UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA UNIVERSITARIA DE
INGENIERÍA TÉCNICA DE
TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

Open Technologies for Prototyping the Internet of Things

Autor
Mª Jesús Martínez Abascal

Tutor
José Fernán Martínez Ortega
Dr. Ingeniero de Telecomunicación

Septiembre 2013

E.U.I.T. TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA
PLAN 2000

TEMA: Internet de las Cosas

TÍTULO: Open Technologies for Prototyping the Internet of Things

AUTOR: Mª Jesús Martínez Abascal

TUTOR: José Fernán Martínez Ortega **Vº Bº.**

DEPARTAMENTO: DIATEL

Miembros del Tribunal Calificador:

PRESIDENTE: Francisco Aznar Ballesta

VOCAL: José Fernán Martínez Ortega

VOCAL SECRETARIO: Vicente Hernández Díaz

DIRECTOR:

Fecha de lectura:

Calificación: **El Secretario,**

RESUMEN DEL PROYECTO:

La memoria de este Proyecto Fin de Carrera comienza con el análisis de plataformas hardware de tipo open source que estimulen el desarrollo creativo de aplicaciones para la Internet de las Cosas, así como plataformas web para el almacenamiento masivo de datos de cualquier dispositivo con conexión a la Internet.

La memoria culmina con el estudio exhaustivo de la propuesta y la especificación de una arquitectura software orientada a servicios para la gestión de sistemas empotrados, que permita la comunicación entre los dispositivos de la red y la transmisión de datos a sistemas externos, así como facilitar el desarrollo de aplicaciones a los programadores mediante la abstracción de la complejidad del hardware.

Este Proyecto Fin de Carrera se enmarca dentro del Proyecto Europeo de Investigación Web of Objects con el sello ITEA2, que potencia la investigación y el desarrollo de las tecnologías más punteras relacionadas con los Sistemas y los Servicios Software.

ABSTRACT

The worldwide "hyper-connection" of any object around us is the challenge that promises to cover the paradigm of the Internet of Things. If the Internet has colonized the daily life of more than 2000 million[1] people around the globe, the Internet of Things faces of connecting more than 100000 million[2] "things" by 2020.

The underlying Internet of Things' technologies are the cornerstone that promises to solve interrelated global problems such as exponential population growth, energy management in cities, and environmental sustainability in the average and long term.

On the one hand, this Project has the goal of knowledge acquisition about prototyping technologies available in the market for the Internet of Things. On the other hand, the Project focuses on the development of a system for devices management within a Wireless Sensor and Actuator Network to offer some services accessible from the Internet.

To accomplish the objectives, the Project will begin with a detailed analysis of various "open source" hardware platforms to encourage creative development of applications, and automatically extract information from the environment around them for transmission to external systems. In addition, web platforms that enable mass storage with the philosophy of the Internet of Things will be studied.

The project will culminate in the proposal and specification of a service-oriented software architecture for embedded systems that allows communication between devices on the network, and the data transmission to external systems. Furthermore, it abstracts the complexities of hardware to application developers.

1 ITU Report, "TRENDS IN TELECOMMUNICATION REFORM 2013"
2 Michael Nelson, IBM IT company director

RESUMEN

La “híper-conexión” a nivel mundial de cualquier objeto que nos rodea es el desafío al que promete dar cobertura el paradigma de la Internet de las Cosas. Si la Internet ha colonizado el día a día de más de 2000 millones[1] de personas en todo el planeta, la Internet de las Cosas plantea el reto de conectar a más de 100000 millones[2] de “cosas” para el año 2020.

Las tecnologías subyacentes de la Internet de las Cosas son la piedra angular que prometen dar solución a problemas globales interrelacionados como el crecimiento exponencial de la población, la gestión de la energía en las ciudades o la sostenibilidad del medioambiente a largo plazo.

Este Proyecto Fin de Carrera tiene como principales objetivos por un lado, la adquisición de conocimientos acerca de las tecnologías para prototipos disponibles en el mercado para la Internet de las Cosas, y por otro lado el desarrollo de un sistema para la gestión de dispositivos de una red inalámbrica de sensores que ofrezcan unos servicios accesibles desde la Internet.

Con el fin de abordar los objetivos marcados, el proyecto comenzará con un análisis detallado de varias plataformas hardware de tipo “*open source*” que estimulen el desarrollo creativo de aplicaciones y que permitan extraer de forma automática información del medio que les rodea para transmitirlo a sistemas externos para su posterior procesamiento. Por otro lado, se estudiarán plataformas web identificadas con la filosofía de la Internet de las Cosas que permitan el almacenamiento masivo de datos que diferentes plataformas hardware transfieren a través de la Internet.

El Proyecto culminará con la propuesta y la especificación una arquitectura software orientada a servicios para sistemas empotrados que permita la comunicación entre los dispositivos de la red y la transmisión de datos a sistemas externos, así como facilitar el desarrollo de aplicaciones a los programadores mediante la abstracción de la complejidad del hardware.

1 Estudio patrocinado por la UIT, “TENDENCIAS EN LAS REFORMAS DE TELECOMUNICACIONES 2013”
2 Michael Nelson, presidente de la compañía IBM IT

ACKNOWLEDGEMENTS

Special thanks to my family and my friends who have provided so much support and love and from whom I have learned so much.

Special thanks also to the researcher engineers Pedro Castillejo, Sandra Cuerva and Jesús Rodríguez for their advices, uninhibited feedback and useful suggestions, and to my "partner", Néstor Lucas who with I kept long "prototyping" discussions throughout the Project development process.

I would also thank to the doctors Vicente Hernández and José Fernán Martínez for their patience and giving me the opportunity to make my Final Project Dissertation in the emerging field of the Internet of Things.

Finally, I would like to thank many more individuals who directly or indirectly supported this effort.

CONTENTS

PART II: CASE STUDY

PART III: CONCLUSIONS AND FUTURE WORKS

ANNEXES

TABLE OF FIGURES

TABLE OF CHARTS

ACRONYMS

AC	**Alternating Current**
ADC	Analog-to-Digital Converter
API	Application Programming Interface
API	Application Programming Interface
BSD	Berkeley Software Distribution
CAN	Controller Area Network
CSI	Camera Serial Interface
CSV	Comma Separated Values
Customer	Event consumer
DAC	Digital-to-Analog Converter
DC	Direct Current
DDR3 SDRAM	Double Data Rate type Three Synchronous Dynamic Random Access Memory
DIP	Dual In-line Package
DRAM	Dynamic Random Access Memory
DSI	Display Serial Interface
EDA	Event-Driven Architecture
eMMC	Enhanced Module Management Controller
GB	Gigabyte
GHz	Gigahertz
GNU	General Public License
GPIO	General Purpose Input Output
HDMI	High-Definition Multimedia Interface
HDMI	High-Definition Multimedia Interface
HTTP	Hyper Text Transfer Protocol
I/O	Input/Output
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environmet
IoT	Internet of Things
JSON	JavaScript Object Notation
JTAG	Join Test Action Group
KB	Kilobyte
LCD	Lyquid Crystal Display
LDO	Low Drop Out regulator
LED	Light Emitting Diode
MB	Megabyte
MHz	Megahertz
MIPS	Million Instructions Per Second
MMC	Multi Media Card
nSOM	nano Service-Oriented Middleware
NTSC	National Television System Committee
OLED	Organic Light-Emitting Diode
PAL	Phase Alternating Line
PCB	Printed Circuit Board
PMIC	Power Management Integrated Circuit
PNG	Portable Network Graphic
PWM	Pulse-width modulation
RAM	Random Access Memory
RCA	Radio Corporation of America

REST	Representational State Transfer
RFID	Radio Frequecy Identification
RGB	Red Green Blue
SDK	Software Development Kit
SOA	Service-Oriented Architecture
SPI	Serial Peripheral Interface Bus
UART	Universal Asynchronous Receiver/Transmitter
URI	Uniform Resource Identifier
USART	Universal Synchronous Receiver/Transmitter
USB	Universal Serial Bus
uSD	Micro Secure Digital
WiFi	Wireless Fidelity
WSAN	Wireless Sensor and Actuator Network
WSAN	Wireless Sensor and/or Actuator Network
XML	eXtensible Markup Language

CHAPTER 1: INTRODUCTION

1.1. Defining the Internet of Things

The phrase "Internet of Things" (IoT) was introduced in 1999 by Kevin Ashton, executive director of the Auto-ID Center, and was initially linked to Radio Frequency Identification (RFID) technology (Vermesan, et al., 2010). The research report published by International Telecommunication Union (ITU) in 2005 contributed significantly to the proliferation of IoT papers in recent years. Since then, several funded research reports, white papers and individuals have provided their own vision about IoT paradigm.

IoT concept is evolving and the vision of what exactly the IoT will be is still emerging. European Research Cluster on the Internet of Things (IERC) provides a definition which covers the many facets of IoT concepts around the world:

> *"Internet of Things is an integrated part of Future Internet including existing and evolving Internet and network developments and could be conceptually defined as a dynamic global network infrastructure with self configuring capabilities based on standard and interoperable communication protocols where physical and virtual "things" have identities, physical attributes, and virtual personalities, use intelligent interfaces, and are seamlessly integrated into the information network.*
>
> *In the IoT, "smart things/objects" are expected to become active participants in business, Information and social processes where they are enabled to interact and communicate among themselves and with the environment by exchanging data and information "sensed" about the environment, while reacting autonomously to the "real/physical world"* ***events*** *and influencing it by running processes that* ***trigger actions*** *and create services with or without direct human intervention."*

Estimates suggest that in 5 to 10 years there will be 100 billion devices connected to the Internet (Michael Nelson, IBM IT director). The first direct consequence will be the generation of huge quantities of data in future networks so the design of large-scale IoT deployment is required.

IoT will be composed of multiple heterogeneous systems (computers networks, RFID systems, sensor networks, etc.) which transmit data among themselves. IoT's dynamism implies that a "thing" can be connect or disconnect itself to the network anytime, anyplace, with anything and anyone (ITU, 2005). Each "thing" will share information with specific data formats and send it with specific protocols.

RFID systems and Sensor and/or Actuator networks will constitute the backbone of IoT. Hundreds of devices will be part of these systems. Manual management of these devices will not be viable in the future. These systems are required to set automatically their parameters. Many specific words associated to automatic management can be found in the literature as "self-configuration", "self-optimization", "self-healing" or "self-protection" (Debortoli, et al., 2012).

The figure below represents a generic wireless sensor network with the basic elements that characterize it as own nodes and a sink where there are all communications to and from the exterior of the wireless sensor network. Most common wireless communication at a physical level among nodes is the 802.15.4 Protocol.

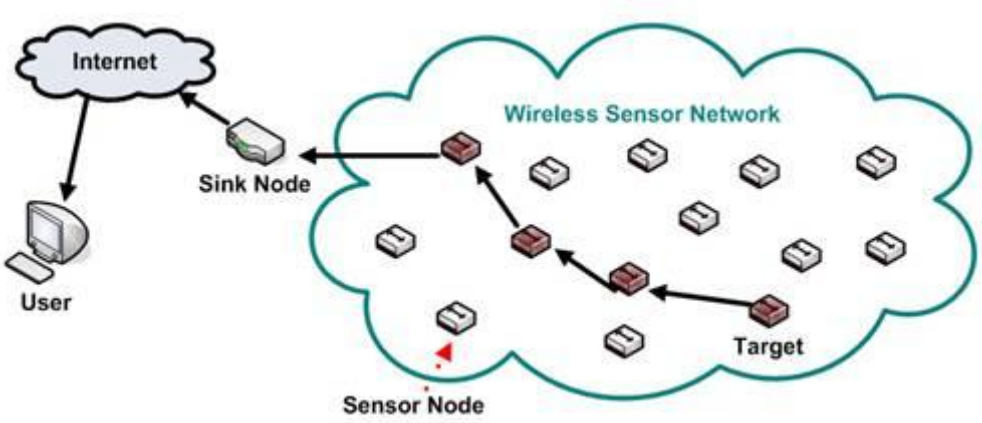


Figure 1: Generic Wireless Sensor and Actuator Network (Postech, 2013)

1.2. What are "Things"?

The "things" are the end points of the IoT and can be any "world object" equipped with appropriate technologies such as RFID tags, sensors or actuators. Due to micro systems integration advances most of these electronic devices are composed of different units as wireless identification technologies, memory, antennas, etc. They can be wired or wirelessly networked and interconnected and are able to sense, communicate, interact and exchange data to the Internet (Vermesan, et al., 2010).

IoT scenarios envision "things" as nodes of the Internet; to make them addressable they will be unique identifiable with Internet Protocol version 6 (IPv6) directions. However many proposals use a gateway with addressable direction between sensor and/or actuator networks and the Internet.

1.3. Project Goals

Learning about the underlying technologies associated with the Internet of things available in the market for the development of applications and prototypes. Furthermore, developing skills related to design and specification in a real scenario. The core objectives are the following:

- Looking for "open source" hardware platforms that could be applied to Internet of Things' scenarios.
- Looking for cloud platforms for mass storage of data sent by hardware platforms.
- Studying and developing a real scenario of the Internet of Things.

1.4. Project Context

This Final Project Dissertation was developed in the context of the research center Centro de Investigación en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad (CITSEM) of the Universidad Politécnica de Madrid. The Project is encased in the following European research project:

WoO: Web of Objects. ITEA2 project referenced as TSI-020400-2011-29.

It is funded by the subprogram Avanza Competitividad from The Spanish Ministry of Industry, Tourism and Commerce. It is supported by ITEA2, a program to stimulate researching which belongs to the European organism EUREKA.

1.5. Project Organization

The approach followed in this work has been structured in three main parts.

The **first part** groups the chapter 2 and chapter 3. The chapter 2 gives detailed information of the main "open source" hardware platforms focusing on software and hardware resources that could be applied to an Internet of Things' scenario. Chapter 3 deals with some cloud platforms that follow the Internet of Things' philosophy which gives tools for mass storage and remote manipulating of data.

The **second part** focuses in the study a scenario of the Internet of Things. It is encased in a Wireless Sensor and Actuator Network (WSAN) which provides some services accessible from the Internet. Chapter 4 is devoted to present an exhausted study of nSOM software architecture which makes possible the communication between the WSAN and the Internet. Chapter 5 describes the scenario deployed to check the architecture, and the main use cases.

The **third part** deals with the conclusions along with the future works regarding this work.

Finally, two **annexes** are provided; the first exposes a comparison of "open source" hardware platform, and the second another comparison regarding to the "HELLO World" program of every hardware platform analyzed.

PART I: INTERNET OF THINGS' TECHNOLOGIES

CHAPTER 2: **OPEN HARDWARE PLATFORMS**

CHAPTER 3: **CLOUD PLATFORMS**

CHAPTER 2: OPEN HARDWARE PLATFORMS

2.1. Introduction

In our daily life, we are surrounded of many "things" or "objects", and they may be electronic devices or not. Electronic devices are varied and wide, and they can be more or less complex. They can be interpreted as computer systems which integrate hardware and software. Devices are distinguished by their intended purposes, which condition their hardware and software design (e.g. industrial machines, smart phones, desktop computers, etc.). However, all of them are governed under the same principles of operation. All have a processor – the heart of the hardware platform – a memory to store data, and other peripherals. Regarding the processor, we can find two types, microcontrollers and microprocessors; in a holistic point of view, microcontrollers are those which in a single chip or "microchip" integrate the embedded system and peripherals, such as RAM and ROM memories, among others; microprocessors are "microcontrollers" without integrated peripherals in their microchip. Microcontrollers often are focused on a specific task and can run only one application indefinitely (e.g. Arduino platforms which have not an operating system by default); microprocessors are usually integrated in more complex systems and can run several applications (e.g. smart phones with the operating system Android, or desktop computers with operating systems such as Windows, Linux or Mac).

The figure below shows three software models, which can be applied to any electronic device. Therefore, their components, both software and hardware can be grouped with this modular representation.

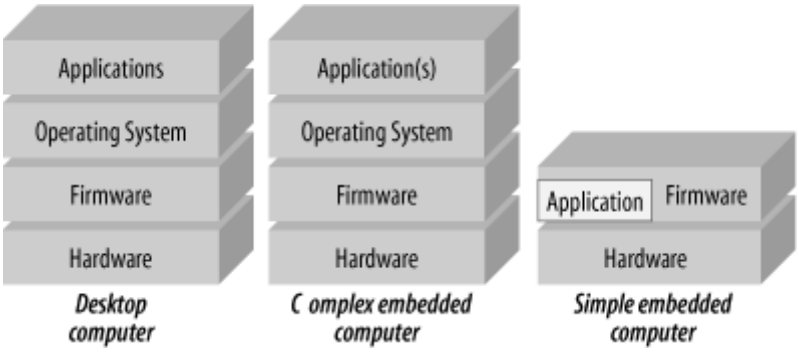


Figure 2: Generic software stack (Catsoulis, 2005)

According to the previous figure, the more complex is a system, the more software layers it has. Note that the three models shown have the "firmware" layer. Firmware is a group of programs run by the processor when the computer system is powered up.

This chapter covers an analysis of "incomplete" hardware platforms, that is to say, those which intended purpose that not satisfy any specific need. However, these "incomplete" platforms give to developers the opportunity for exploring and customizing solutions not found in the market.

Wireless sensor motes are exceptions of this study since they are designed to be used in specific scenarios regarding Wireless Sensor and Actuator Networks (WSAN). In addition, I will focus on hardware platforms that follow the "open-source" philosophy.

I have classified the analized platforms in two main parts; single-board microcontrollers which would be associated to "Simple embedded computer" of the above figure; and single-board computers featured by a microprocessor which could be associated to "Desktop computer" of the figure. However, these last platforms are not desktop computers at all, but they are a mix between smart phones and laptops for prototyping purposes.

2.2. Single-board microcontrollers

2.2.1. Arduino

2.2.1.1. Introduction

Arduino Project started in 2005 and was developed at the Design Institute of Ivrea (Italy) for educational purposes. Arduino Project combines the Arduino Integrated Development Environmet (IDE) and the hardware platform which has their roots in the thesis work that the student Hernando Barragán did on the Wiring platform, an open-source programming framework for microcontrollers (Barragán, 2013) (Arduino, 2013) (Banzi, 2008). Open hardware reference designs are released under a Creative Commons Attribution Share-Alike 2.5 license and the Arduino IDE can be downloaded for free under the General Public License 2 (GNU).

Figure 3: Arduino logo (Ard132)

"Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists and anyone interested in creating interactive objects or environments." (Arduino, 2013)

Arduino family is constituted of sixteen official boards and a broad range of derived boards. In this Final Project Dissertation we will highlight the most representative: Arduino UNO, Arduino DUE and Arduino MEGA 2560.

2.2.1.2. Hardware blocks

The physical computing platform is a printed circuit board (PCB) or Input / Output board built upon Atmel 8-bit AVR and 32-bit Atmel ARM microcontrollers, it depends on the version of board as it will be shown in the next sections. Boards are featured by several integrated blocks as it shown in the following figure.

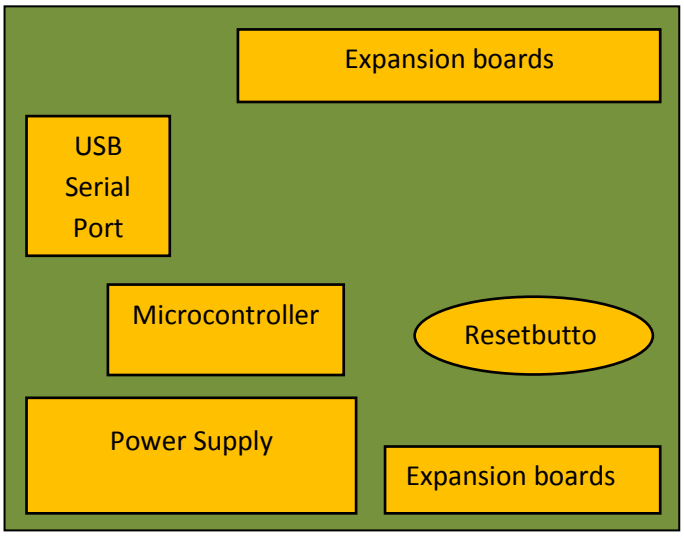


Figure 4: Arduino hardware blocks

- **Processor** is the brain of the board which characterizes processing capabilities of the board such as memory or clock rate. I/O Boards are built upon Atmel 8-bit AVR and 32-bit Atmel ARM microcontrollers depending on the Arduino's version. The microcontroller is pre-programmed with a bootloader that simplifies uploading programs to the on-chip flash memory and is no need of an external programmer.
- **Serial port** is usually to communicate the Arduino and the computer while compiling and uploading code programs to it or connecting with other serial devices. Arduino boards offer several serial communications protocols such as I2C (Inter-Integrated Circuit), SPI (Serial Peripheral Interface Bus), UART (Universal Asynchronous Receiver/Transmitter) or USART (Universal Synchronous Receiver/Transmitter) that are used for connecting add-on modules or shields. There are a wide variety of shields compatible with Arduino boards such as Bluetooth, Ethernet, WiFi, GSM/GPRS which permit connectivity and gives the possibility of being used in the Internet of Things for transmitting or receiving data among a variety of networks.
- **Expansion boards** encapsulate the typical digital inputs / outputs, analogical that combined with serial communication protocols makes possible connecting external shields, sensors or other complementary circuitry.
- In order to feed the board, a **power supply** connector is provided.

2.2.1.3. Software stack

Arduino IDE is an application which can be installed in Windows, Linux and Macintosh and gives to the user a development environment for coding the programs that are uploaded to the board. Traditional boards were programmed over an RS-232 serial connection but current allow over USB using a USB-to-serial adapter such as FT232. These programs are called "sketches".

Arduino provides a wide range of libraries which make easy to manage the different inputs / outputs or shields. The following figure shows a software stack with some of most common libraries.

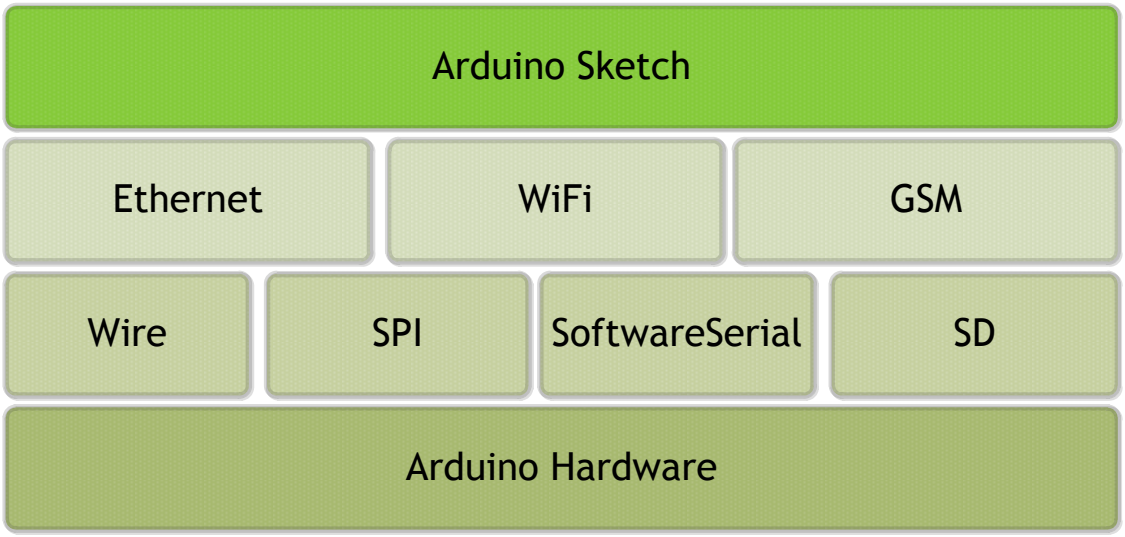


Figure 5: Arduino software stack

Arduino IDE provides standard libraries to manipulate hardware and data easy-to-use. Some of the libraries shown in the figure above are for handling serial communication protocols such as I2C associated to library "Wire" or for setting connectivity like "Ethernet", "GSM" or "WiFi".

2.2.1.4. Arduino UNO

Arduino UNO was launched on September 2011 (Wheat, 2011) and is intended to be the "standard" Arduino and the reference model to be based the upcoming boards. It should be highlighted that the main feature is its USB-to-serial converter and removable microcontroller. The following technical features correspond to Arduino UNO Revision 3. Arduino UNO is similar to Arduino Leonardo, however this last one has built-in USB communication, as a result, there is no need for a secondary processor and the board is slightly cheaper.

Figure 6: Arduino UNO versus Arduino Leonardo (Arduino, 2013)

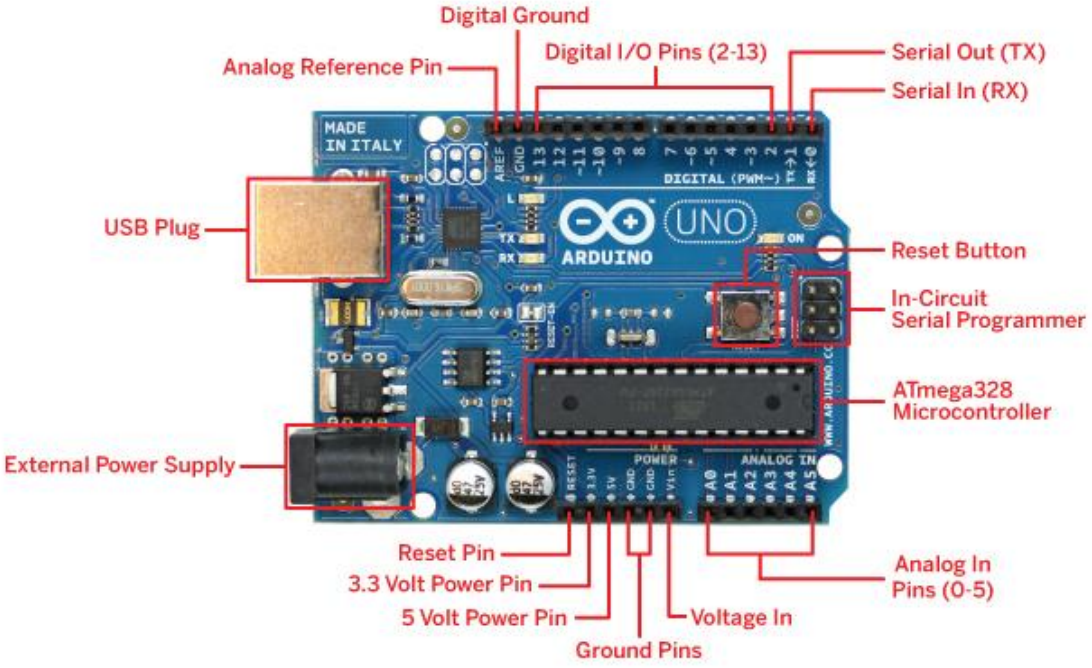


Figure 7: Arduino UNO detailed pinout (ArduinoBrasil, 2013)

	Arduino UNO	Arduino Leonardo
Processor and memory		
Microcontroller	ATmega328 (8-bit)	ATmega32u4 (8-bit)
Flash Memory	32 KB	32 KB
SRAM	2 KB	2.5 KB
EEPROM	1 KB	1 KB
Clock speed	16 MHz	16 MHz
Inputs / Outputs		
Digital I/O pins	14 (6 for PWM output)	20 (7 for PWM output)
Analog input pins	6	12
Communication	UART, I2C, SPI, USB, 6 PWM	UART, I2C, SPI, USB, 7 PWM
External interrupts	2	2
Others		
Size	7 x 5 cm	7 x 5 cm
Price	20 € (1-9 units)	18 € (1-9 units)
Reset	Hardware / Software	Hardware / Software
Schematic & Reference Design	http://arduino.cc/en/Main/ArduinoBoardUno	http://arduino.cc/en/Main/ArduinoBoardLeonardo

Chart 1: Arduino UNO and Arduino Leonardo technical features

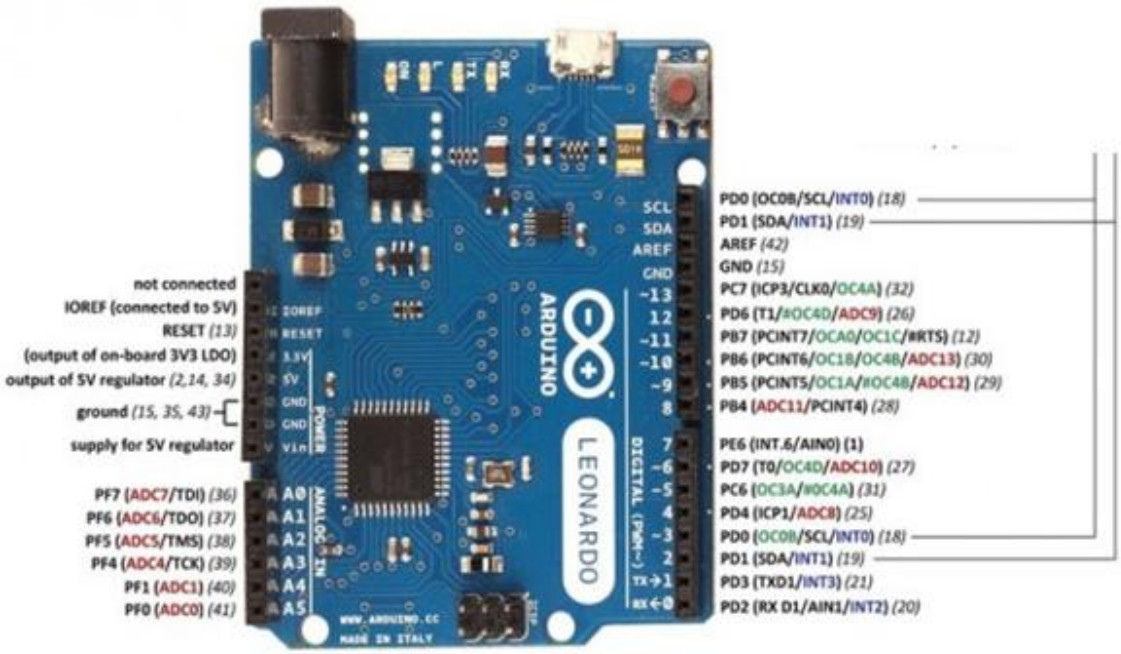


Figure 8: Arduino Leonardo detailed pinout (ElectroSchematics, 2013)

2.2.1.5. Arduino DUE

It is the first Arduino platform developed based on 32-bit microcontroller which empowers computing processing capabilities. In addition, it differs from other boards in that it is powered with 3.3 V and not with 5 V, as usual. The board has one micro-USB B connector for connecting external USB peripherals (e.g. keyboards, mouse, etc.), and another for debugging purposes. It should be noted that the size of the board is bigger than Arduino Uno board is.

Figure 9: Arduino DUE platform (Arduino, 2013)

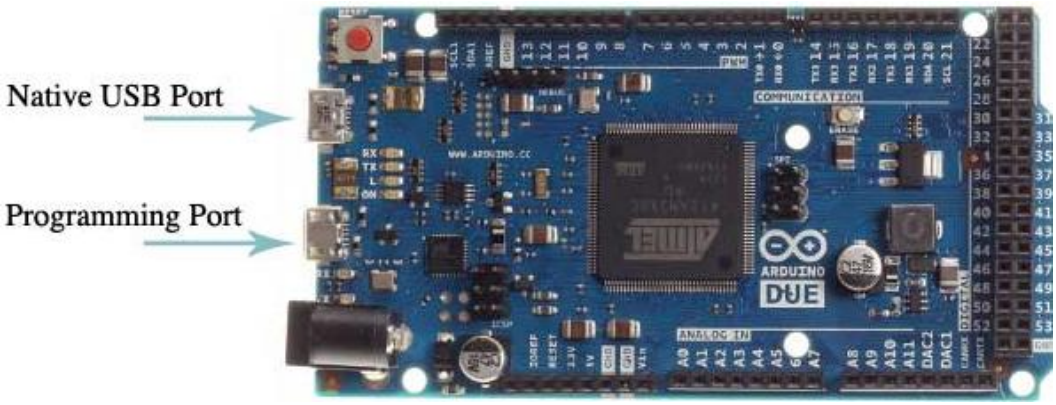


Figure 10: Arduino DUE programming ports (trastejant, 2013)

Processor and memory	
Microcontroller	Atmel SAM3X8E ARM Cortex-M3 CPU (32-bit)
Flash Memory	512 KB
SRAM	96 KB
Clock speed	84 MHz
Inputs / Outputs	
Digital I/O pins	54 (12 for PWM output)
Analog pins	12 (input), 2 DAC
Communication	UART, 3 USARTs, 2 I2C, SPI, USB, 12 PWM
External interrupts	2
Others	
Size	10 x 5 cm
Price	39 € (1 – 9 units)
Reset	Hardware / Software / Erase
Schematic & Reference Design	http://arduino.cc/en/Main/ArduinoBoardDue

Chart 2: Arduino DUE technical features

2.2.2. Netduino

2.2.2.1. Introduction

"*Netduino is an open source electronics platform using the .NET Micro Framework. It Features a 32-bit microcontroller and a rich development environment and is suitable for engineers and hobbyists alike*" (netduino, 2013) (Pfister, 2011). Netduino firmware is provided under Apache 2.0 and BSD open source licenses and the hardware design files – schematics and layout files – under Creative Commons-Atribution license. Netduino family is formed of three electronic boards: Netduino 2, Netduino Plus 2, and Netduino GO.

Figure 11: Netduino logo (netduino, 2013)

2.2.2.2. Hardware blocks

The physical computing platform is an Input / Output board built upon STMicro STM32F2 and STM32F4 32-bit microcontrollers; it depends on the version of board as it will be shown in the

next sections. Boards are featured by several integrated blocks as it shown in the following figure. It should be noted that hardware blocks, even the distribution of these ones are very similar to the Arduino UNO platform.

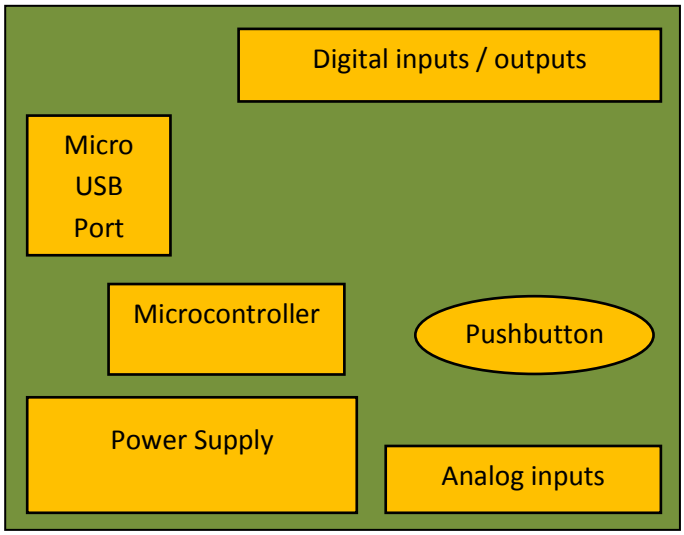


Figure 12: Netduino hardware blocks

- **Processor** – determines the processing capabilities of the platform and contains the code storage and RAM used by the Netduino app. It handles the connections to the blue pin headers (shown in the figures of Netduino platforms in the following sections) to add complementary circuitry and / or expansion shields.
- **Micro USB port** – is used to connect the Netduino to the computer's USB port and deploying apps and debugging them. This port can also be used to make the Netduino a USB device such as keyboard, mouse, etc.
- **Analog input headers** – let the developer to plug sensors such as light, temperature, motion, pressure, etc.
- **Digital Input / Output headers** - serve for many general purposes regarding plugging on / off inputs such as switches, pushbuttons or on / off sensors, plugging on / off outputs such as LEDs or relay switches. Several digital I / O headers can communicate using standard serial communication protocols such as I2C, SPI or UART and other admit PWM mode.
- **Power supply** – consists of a power barrel jack for powering the board by AC-to-DC power, and power regulation circuitry to homogenize the incoming higher voltage into the 3.3V needed by the microcontroller. The board can also be powered from the computer over the Micro USB port.
- **Pushbutton** – resets the Netduino platforms by default; as a result, the deployed app is restarted. Alternatively, it can be used for other purposes.

- **Other notably blocks** – include an erase pad located underneath pin 0 which serves for flashing the Netduino firmware from scratch; this let the developer to install other operating systems or prepare the Netduino platform as a microcontroller development board. In addition, boards are equipped with indicators for showing when the platform is powered and when it is powering on or off; one white LED and another blue LED, respectively. The last one can be programmed in the Netduino app.

2.2.2.3. Software stack

Netduino projects are programmed under SPOT Smart Watch, a runtime programming introduced by Microsoft in 2004 which derives of desktop .NET in order to program microcontrollers in C# programming language. Developers' code programs are called "apps". C# allows developers focus on the apps, while letting the underlying framework take care of all the low-level details. The underlying framework is .NET Micro Framework which provides a powerful set of features such as events, threading, and line-by-line debugging.

Coding Netduino apps requires three software packages on the host computer with Windows XP or Windows 7:

1. The Visual Studio development environment – is a code editor for writing and debugging apps. There are both free and paid versions available.
2. The .NET Micro Framework software development kit (SDK) – "is a powerful and flexible platform for rapidly creating embedded device firmware with Microsoft Visual Studio" (Microsoft, 2013).
3. The Netduino SDK – includes project templates that make it easy to get started with Netduino, USB drivers for Netduino, and other Netduino-specific tools.

The Mono project (Mono, 2013) provides an alternative for using .NET programming tools on a Mac or Linux computer.

The following figure shows the high level software and hardware stack of Netduino platform.

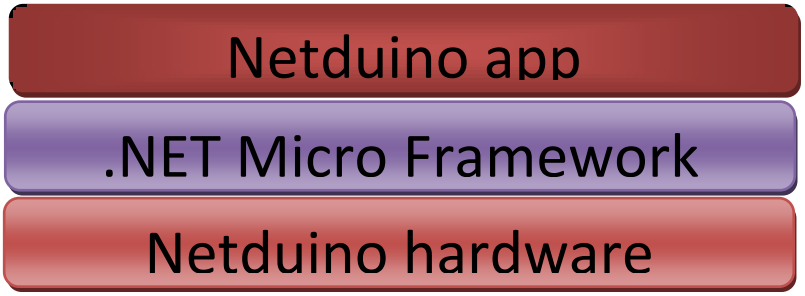


Figure 13: Netduino software stack

Although .NET Micro Framework architecture is not a goal of this Final Project Dissertation, a figure below shows the most representative layers in order to get more in detail of .NET Micro Framework layers. The user applications would be Netduino apps and the Processor and peripherals would correspond with Netduino hardware.

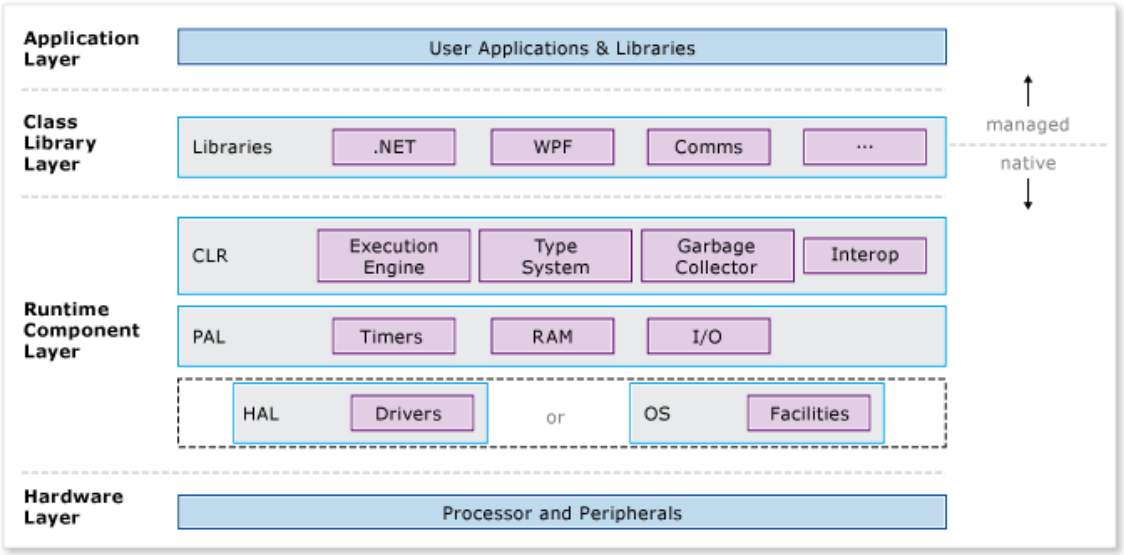


Figure 14: .NET Micro Framework software and hardware architecture (Microsoft, 2013)

2.2.2.4. Netduino 2 versus Netduino plus 2

Netduino 2 is the entry-level board in the Netduino family. Netduino Plus 2 is an enhanced version of Netduino 2 adding storage and networking features. Netduino Plus incorporates an Ethernet jack and a MicroSD slot which let persistent storage.

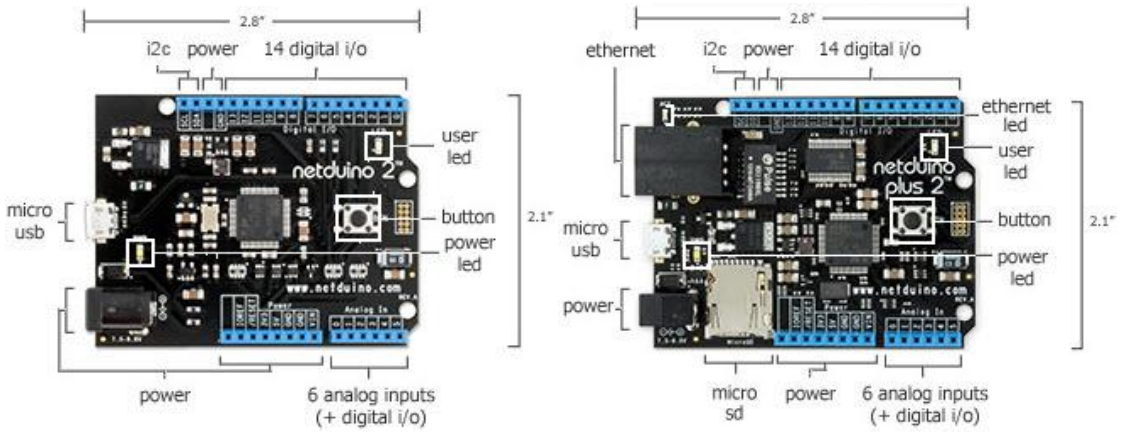


Figure 15: Netduino 2 (left) and Netduino Plus 2 (right) detailed pinout (netduino, 2013)

	Netduino 2	Netduino Plus 2
Processor and memory		
Microcontroller	STMicro STM32F2	STMicro STM32F4
Flash Memory	192 KB	384 KB
SRAM	60 KB	100 KB
Clock speed	120 MHz (Cortex-M3)	
Inputs / Outputs		
Digital I/O pins	22	
Analog	6 ADC channels (12 bit)	
Communication	6PWM, 4 UART, I2C, SPI	6PWM, 4 UART, I2C, SPI, Ethernet
Storage	Add-on: sd shields (up to 2 GB)	Micro sd (up to 2 GB)
Others		
Size	2.1" x 2.8"	
Price	$ 34.95	$ 59.95
Schematic & Reference Design	http://www.netduino.com/downloads/	
Arduino shield compatibility	Works with most Arduino shields (some requires .net mf drivers)	

Chart 3: technical features - Netduino 2 versus Netduino Plus 2

2.2.2.5. Netduino GO

Netduino GO! Goals to be a board in which plug a wide range of modules, while isolates the developer the complexity of electronic projects. It is the base for building electronic projects upon blocks of modules which expand its functionality, and it is compatible with gobus modules (blue headers).

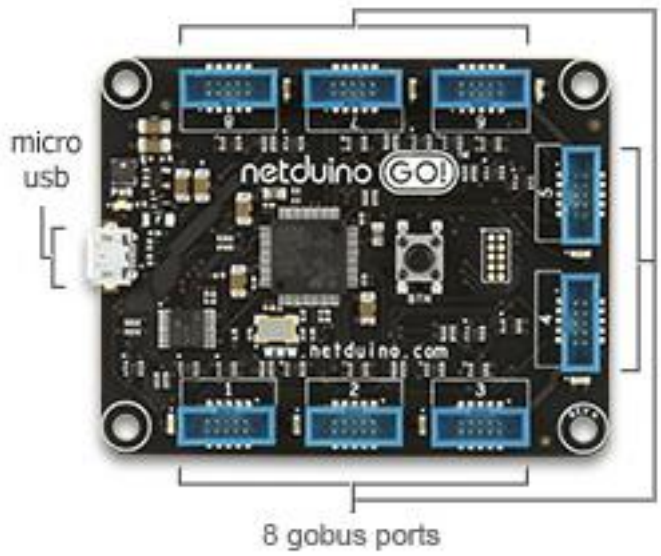


Figure 16: Netduino GO detailed pinout (netduino, 2013)

Processor and memory	
Microcontroller	STMicro STM32F4 (32 bits)
Code storage	384 KB
SRAM	100 KB
Clock speed	120 MHz (Cortex-M3)
Inputs / Outputs	
Digital I/O pins	Add-on: gobus i/o modules (GPIO)
Analog	Add-on: analog gobus modules
Communication	Add-on: gobus i/o modules (PWM, UART, SPI, and more)
Gobus ports	8
Storage	Add-on: SD card gobus module
Others	
Size	2.1" x 2.7"
Price	$ 99.95
Schematic & Reference Design	http://www.netduino.com/downloads/
Arduino shield compatibility	Works with most Arduino shields (some requires .net mf drivers)

Chart 4: Netduino GO technical features

2.2.3. Wireless sensor motes

2.2.3.1. Introduction

Motes are particular single-board microcontrollers for developing applications in Wireless Sensor and Actuator Networks (WSAN). In contrast of the previous studied platforms, they integrate into their boards a wireless module communication and a battery for placing them "anywhere". However, these types of platforms are tending to integrate "blocks" of hardware in their single board to add extra modules circuits in order to expand their functionalities (e.g. Waspmote Pro accepts several modules such as GPS, Bluetooth, GPRS, 3G, etc.).

The figure above shows the basic hardware blocks of a generic wireless sensor mote.

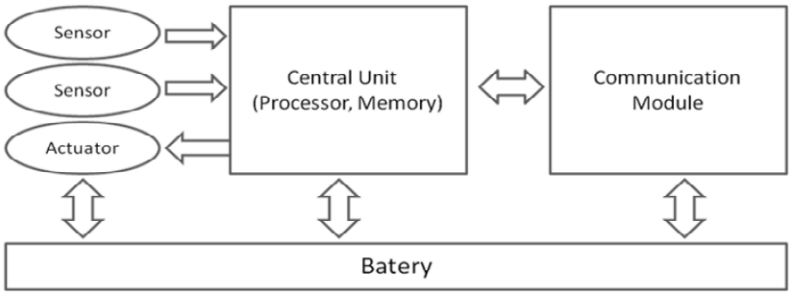


Figure 17: Hardware blocks of a generic wireless sensor mote (Martínez Ortega, et al., 2010)

2.2.3.2. SunSPOT

Project Sun SPOT (Sun Small Programmable Object Technology) was born in Sun Labs in 2003 within Sun Microsystems Company with the objective to explore wireless sensor networks and embedded systems (SunSPOTworld, 2013). In 2010 Sun Microsystems was purchased by Oracle Corporation which caused to stop the platform development by mid-2011. However, Sun SPOT motes are still widely used in researching and education due to their unique features – Sun SPOT applications are programmed in Java language which makes it perfect for rapid development software.

Figure 18: Sun SPOT logo (Oracle, 2013)

Although the Sun SPOT technology's maintenance was stopped, in July 2013 Oracle re-launched the official forum again. On the other hand, the old leading researchers of the Sun SPOT project launched another non-official forum in order to keep alive Sun SPOT innovation platform.

2.3.3.2.1. Software stack

As mentioned above, Sun SPOT motes software development is in Java, unlike other embedded systems which are in C/C++ or C-variants. Sun Microsystems developed an operating system for the platform called Squawk which is a Java Virtual Machine adapted for the platform. Squawk offers a user friendly application programming interface (API) in Java in order to abstract complex hardware details like access to the wireless communication module and the different sensors. Plus, Squawk allows to use multithreading and multiprocessing so it is possible to run several applications on the same mote. Other interesting features are programming over the air, that is to say, remote programming of the mote and selecting the application in which you want the mote to run (for instance, one application could be reading temperature and other application making ring noise, so you can select remotely if you can run one application or two at the same time; even more, you can send in remotely form a third application and make it run with the other at the same time). Regarding hardware technical features, RAM and ROM memory values are particularly high compared to other embedded systems.

Next figure exposes the software stack of a free-range Sun SPOT mote.

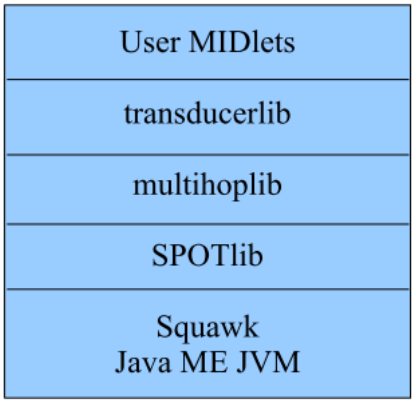


Figure 19: Free-range SPOT mote software stack (Sun-Labs, 2010)

The highest layer corresponds to SPOT applications written by developers in Java ME programming language. The main applications of this language are technically called "MIDlets". The lowest layer is the Squawk JVM (Java Virtual Machine). Note that Squawk is not an operating system.

In the middle are located the SPOT libraries which are used by developers to build the applications. Sun SPOT classifies its libraries in three levels; the SPOTlib is responsible for accessing to the SPOT device and basic I/O; the multihoplib which provides an API for managing radio protocols such as Radiogram and Radiostream; and the transducerlib which offer classes to handle sensors' data of the board (e.g. temperature sensor).

2.3.3.2.2. Technical features

The next table describes the technical specifications of a free-range SunSPOT mote in detail.

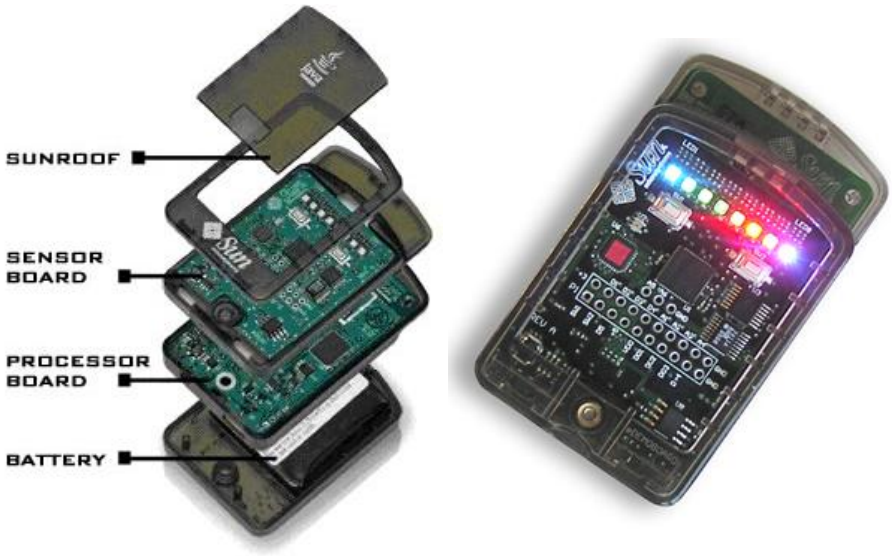


Figure 20: Sun SPOT mote platform (Gouvea, 2010) (ANRG, 2013)

Processor and memory	
Microcontroller	AT91SAM9G20
Flash Memory	8 MB
SRAM	8 KB
EEPROM	4 KB (1 KB reserved)
Clock speed	400 MHz
Inputs / Outputs	
Digital I/O pins	GPIO
Analog	ADC
Communication	UART, I2C, SPI, USB
Sensors	Light, temperature, accelerometer
Radiofrequency	
Transceiver	CC1000
Band	868/916 MHz (MPR400CB), 433 MHz (MPR410CB), 315 MHz (MPR420CB)
Data speed	38.4 Kbps
Modulation	FSK
Range	Outdoors: 152.40 m (MPR400), 304.8m (MPR410 y MPR420)
Others	
Battery	770mAhr Li-Ion Rechargeable Battery
Consumption (idle)	30 mA
Consumption (Rx/Tx)	105.6 mW
Price	630 € per kit (two motes and a base station)

Chart 5: Sun SPOT technical features

2.2.3.3. Waspmote PRO

In February 2013, Libelium Company (Libelium, 2013) launched Waspmote PRO v1.2 which is an evolution of a previous mote Waspmote v1.1 launched in 2009. Waspmote PRO is an open source sensor platform maintained for more than 2,000 developers in order to provide solutions for the Internet of Things and Smart Cities applications (city pollution, emissions from farms and hatcheries, etc.).

Figure 21: libelium logo (Libelium, 2013)

Waspmote PRO mote have been conceived for ultra low power consumption; it claims to be the lowest sensor platform in the market. It is a modular hardware platform and it is compatible with more than 60 sensors (temperature, luminosity, presence, etc.). Its hardware architecture

allows the connection of extra modules which provide to the mote different wireless interfaces. Furthermore, motes can be program wirelessly with OTA commands (Over the Air Programming), and upload sensor data to Internet clouds.

Figure 22: Waspmote platform (lib13)

2.3.3.3.1. Software stack

Libelium provides the Waspmote-IDE for programming their motes, which can be installed on Windows, Linux, and Mac computers. Its compiler is based on the Arduino one, and the organization of the libraries share common features.

Applications are developed upon the Waspmote API which is based in C/C++. The API is divided in two blocks; "cores" which contain the general API and "libraries" which encompasses libraries for the different modules of the mote. The following figure exposes an overview of the software stack on Waspmote PRO mote.

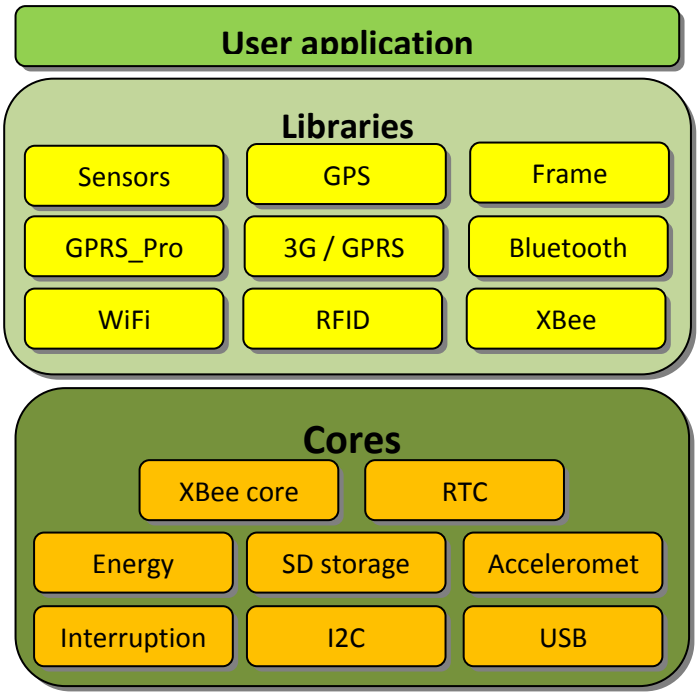


Figure 23: Waspmote PRO mote software stack

2.3.3.3.2. Technical features

Waspmote PRO has been designed to be a modular platform in which can be plugged different modules and a plethora of sensors. Some modules are ZigBee, GSM/GPRS, 3G/GPRS, GPS module or SD memory card. There are a wide range of sensors which can be plugged in the sensor board regarding gases (e.g. carbon monoxide, carbon dioxide, hydrogen, etc.), events (e.g. liquid level, liquid presence, pressure, etc.).

Next figures show the key locations of the hardware and a table with detailed specifications.

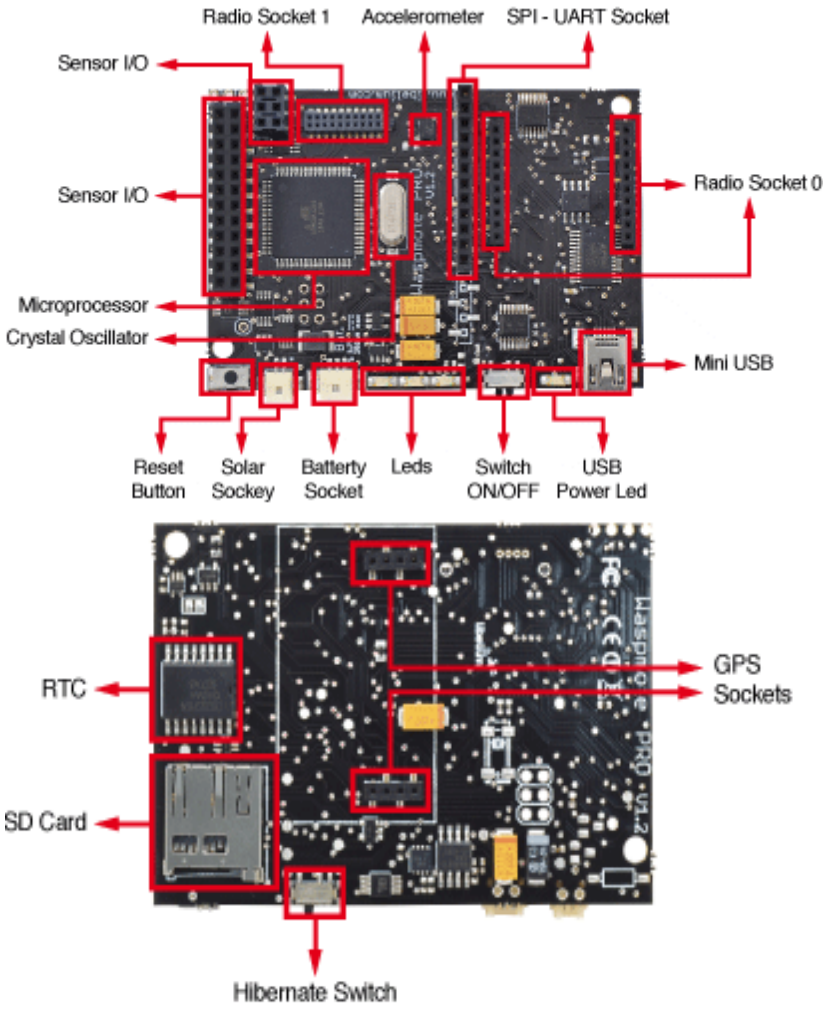


Figure 24: Waspmote detailed pinout (lib131)

Processor and memory	
Microcontroller	ATmega 1281
Flash Memory	128 KB
SRAM	8 KB
EEPROM	4 KB (1 KB reserved)
Clock speed	14 MHz
Inputs / Outputs	
Digital I/O pins	8
Communication	2 UART, I2C, SPI, USB, specific socket for "basic sensor" Temperature, humidity, light
SD card	2 GB
Others	
Power supply	Battery, USB charging, Solar panel charging
Size	73.5 x 51 x 13 mm
Weight	20 g
Price	280 € per kit (Waspmote expansion board, GSM/GPRS module, WiFi module, Bluetooth PRO Module)
Built-in sensors on the board	Temperature, accelerometer

Chart 6: Waspmote technical features (Libelium, 2013)

Waspmote

ON	15mA
Sleep	55µA
Deep Sleep	55µA
Hibernate	0,06µA

Figure 25: Consumption of Waspmote (Libelium, 2013)

Model	Protocol	Frequency	txPower	Sensitivity	Range *
XBee-802.15.4	802.15.4	2.4GHz	1mW	-92dB	500m
XBee-802.15.4-Pro	802.15.4	2.4GHz	100mW	-100dBm	7000m
XBee-ZB	ZigBee-Pro	2.4GHz	2mW	-96dBm	500m
XBee-ZB-Pro	ZigBee-Pro	2.4GHz	50mW	-102dBm	7000m
XBee-868	RF	868MHz	315mW	-112dBm	12km
XBee-900	RF	900MHz	50mW	-100dBm	10Km

Figure 26: Consumption of Waspmote in TX/RX mode and range (Libelium, 2013)

2.3. Single-board computers

2.3.1. Raspberry Pi

2.3.1.1. Introduction

Raspberry Pi Project (Halfacree, et al., 2012) was conceived at Computer Labs of Cambridge University in 2006. Raspberry Pi Foundation launched in 2012 a low-cost System on a Chip platform especially designed for teaching children Physical Computing.

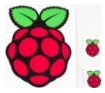


Figure 27: Raspberry Pi logo (RaspberryPi, 2013)

The hardware platform runs Linux-based operating systems such as Raspbian, Adafruit Raspberry Pi Educational Linux, Arch Linux, Xbian, Qton Pi, among others that are coming. The recommended Linux distribution is Raspbian which offers access to several learning environments like "Scratch" for programming games, or "IDLE", a Python editor. The hardware platform's "surname", Pi, is due to Python programming language which is the preferred for accessing the hardware and general purpose applications.

2.3.1.2. Hardware blocks

The single-board computer is designed around the ARM11 architecture. ARM chips provide a variety of architectures with different cores configured for different processing capabilities at different prices. The board is featured by several connectors explained below. The following figure shows a high overview of how these blocks are distributed on the board.

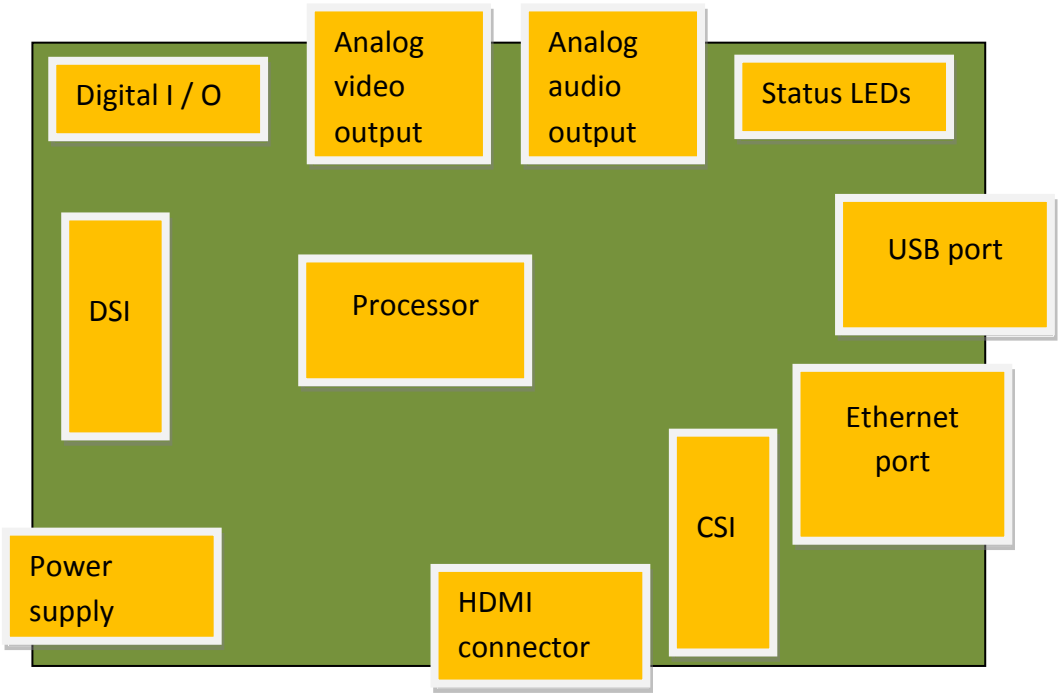


Figure 28: Raspberry Model B hardware blocks

- **Processor** – the brain of the Raspberry Pi is based on a 32-bit, 700 MHz System on a Chip, which is built on the ARM11 architecture. iPhone 3G smart phone and Kindle e-book reader use this processor. The model A has 256 MB of RAM and the Model B 512 MB.
- **External USB port** – the Model A has only one and the Model B has two. It can be used for WiFi connectivity via USB dongle.
- **Ethernet port** – the Model B is the only which has a RJ45 Ethernet port. An option for connecting Model A to the Internet is using its unique USB port for plug-in in USB Ethernet adapter or via WiFi USB dongle.
- **Digital Inputs / Outputs** - serve for many general purposes regarding reading buttons and switches and controlling actuators such as LEDs, relays, or motors.
- **Indicators** – five LEDs marked in the board provide visual feedback. Green ACT LED is illuminated when the SD card is accessed; Red PWR LED is illuminated when power is hooked up to 3.3 V; Green FDX LED lights when network adapter is full duplex; Green LNK LED lights when there is network activity; Yellow 100 LED is illuminated if the network connection is 100 Mbps.
- **Power supply** – Raspberry Pi is fed through a micro USB connector exclusively used for power input.
- **Other connectors** – Camera Serial Interface (CSI) allows a camera module to be connected directly to the board; Display Serial Interface (DSI) for communicating with a LCD or OLED display screen; a standard 3.5 mm mini analog audio jack for using with headphones or unpowered speakers; HDMI for providing digital video and audio output; a standard RCA-type jack for analog video output in composite NTSC or PAL video signals. It is recommended using HDMI connector for video output if having a HDMI television or monitor. HDMI offers much more resolution than composite video out.

2.3.1.3. Technical features

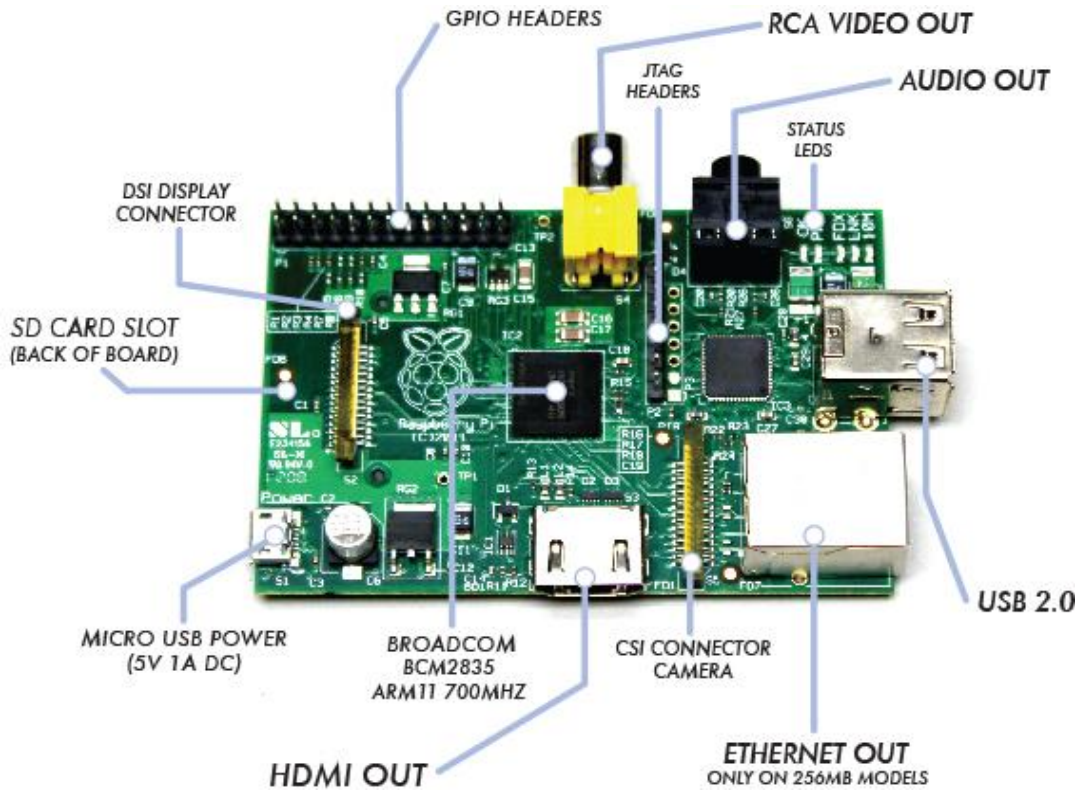


Figure 29: Raspberry Pi detailed pinout (PCMAG, 2013)

	Model A	Model B
Target price	US$25	US35$
System-on-a-chip (SoC)	Broadcom BCM2835 (CPU + GPU SDRAM is a separate chip stacked on top)	
Processor	700 MHz ARM11 ARM1176JZF-S core	
GPU	Broadcom VideoCore IV, OpenGL ES 2.0, OpenVG 1080p30 H.264 high-profile encode/decode	
SDRAM	256 MB (planned with 128 MiB, upgraded to 256 MiB on 29 Feb 2012)	256 MiB (until 15 Oct 2012); 512 MiB (since 15 Oct 2012)
USB 2.0	1 (provided by the BCM2835)	2 (via integrated USB hub)
Video outputs	Composite video | Composite RCA, HDMI (not at the same time)	
Audio outputs	TRS connector | 3.5 mm jack, HDMI	
Audio inputs	None, but a USB mic or sound-card could be added	
Onboard storage	Secure Digital | SD / MMC / SDIO card slot	
Onboard network	None	10/100 wired Ethernet RJ45
Low-level peripherals	General purpose Input/output (GPIO) pins, SPI, I2C, I2S, UART	
Real-time clock	None	
Power ratings (provisional, from alpha board)	500 Ma (2.5 W)	700 Ma (3.5 W)
Power source	5V (DC) via micro USB type B or GPIO header	
Size	85.0 x 56.0 mm (two different boards, measured with callipers)	

Chart 7: Raspberry Pi technical features (RPiHardware, 2013)

2.3.2. BeagleBoard

2.3.2.1. Introduction

BeagleBoard.org Foundation (BeagleBoard, 2013) (Kridner, et al., 2013) was introduced to open-source community in 2008. Currently, its website receives 50,000 visits per month, and is one of the most active collaborative groups in the open-source world.

Figure 30: BeagleBoard logo (BeagleBoard, 2013)

Boards are designed by an employee of Texas Instruments - Gerald Coley – and members of the BeagleBoard.org community. They are not intended to do any particular function, but letting developers to experiment with processor, peripherals, and software.

BeagleBoard.org produces two kinds of boards, BeagleBone and BeagleBoard. BeagleBoard boards are the predecessors in the BeagleBoard family and groups the first of them, BeagleBoard, and the improved one, BeagleBoard-xM. On the other hand, BeagleBone boards are the latest and low cost versions in BeagleBoard family; they are the BeagleBone board, and the younger BeagleBone Black. This section covers the latest versions of each one, BeagleBone Black and BeagleBoard-xM.

In the table below, it can be appreciated the main differences among the boards. The latest board - BeagleBone Black – stands out by its low price compared to the others.

	BeagleBoard	BeagleBoard-xM	BeagleBone	BeagleBone Black
Board				
Quick summary	The original open hardware ARM®-based development board	All features of the original BeagleBoard with extra memory	Low-cost, open-source community platform with plug-in board expansion	Next-generation BeagleBone featuring 1-GHz processor
Memory	256KB L2 cache	512MB DDR2	256MB DDR2	512MB DDR3
Special features	2D/3D graphics acceler-ator, HD video capable, USB powered	1-GHz processing power, four-port hub with 10/100 Ethernet	USB-powered, 10/100 Ethernet, USB JTAG	eMMC, on-board HDMI, USB, Ethernet and HDMI interfaces
Price ($U.S.)	$149	$179	$89	**$45**

Chart 8: Comparison of the BeagleBoard family boards (Kridner, et al., 2013)

2.3.2.2. BeagleBone Black

2.3.2.2.1. Introduction

BeagleBone Black has born to compete with Raspberry Pi both in price, and functionalities. BeagleBone Black differs with its competitor in a better performance of its processor - 1 GHz vs. 720 MHz – and that the kit tools come with all necessary components ready to use, while in Raspberry Pi requires purchasing some core elements that are not included in their price such as SD card, before start prototyping.

2.3.2.2.2. Hardware blocks

Due to the analogy between BeagleBone and BeagleBone Black, it has been considered to establish a comparison between them. The figure above shows that the main differences are the processor, connectors, and user buttons.

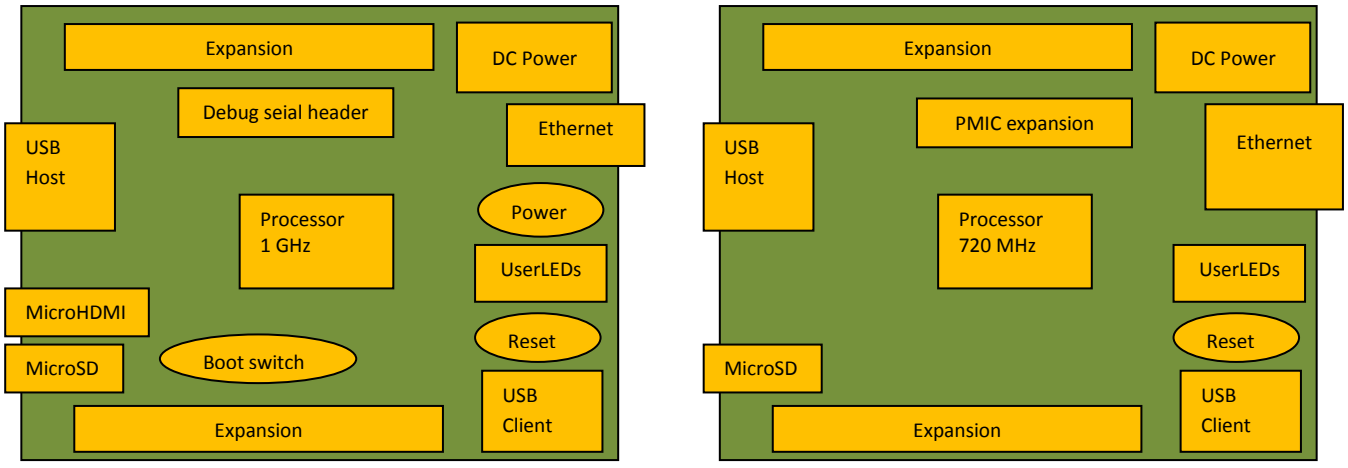


Figure 31: BeagleBone Black (left) and BeagleBone (right) hardware blocks

- **Processor** – BeagleBone Black processor is AM335x 1 GHz ARM Cortex-A8 while BeagleBone is AM335x 1 GHz ARM Cortex-A8. BeagleBone Black processor supports several operating systems such as Linux, Android, Windows Embedded CE, QNX operating systems. It comes with a pre-installed Linux kernel which boots in 10 seconds about. In addition, it offers Cloud9 IDE for developing in a wide range of programming languages such as JavaScript, Java, Ruby, etc.
- **Power supply** – BeagleBone Black board can be powered by the main DC input Power connector of 5 V, or through the USB Client which is connected to a PC. BeagleBone is powered by DC 5V.
- **Indicators** – BeagleBone Black has one blue LED to indicate the board is turned on, other four blue LEDs programmable by users setting GPIO pins, one yellow LED to indicate if the Ethernet connection is 100Mbps, and other green LED which flashes when traffic network. BeagleBone has five green LEDs; one static to indicate the power is on, and other four accessible by the user.

- **Buttons** – while BeagleBone only has a reset button, BeagleBone Black board incorporates a power button for turning on / off the board, a reset button for reinitializing the processor, and a boot switch to force a boot from the microSD.
- **Connectors** – which permit the board to interact with external peripherals like a microHDMI for connecting to a TV or monitor with a microHDMI to HDMI cable, a microSD slot for installing a microSD card, a 10/100 Ethernet to connecting to the LAN, a USB Host which can be connected to WiFi, keyboard, etc., and a serial debug which is the serial debug port to plug a FTDI USB to serial cable.
- **Expansion slots** to give extra functionalities to the board with more than 30 plug-in boards called "capes" such as a BeBoPr 3D Printer cape, a Camera cape, a Weather cape which provides ambient data, Radar cape, etc.

2.3.2.2.3. Technical features

The figure above exposes the key components of BeagleBone Black, and the table compares the technical specifications of BeagleBone and BeagleBone Black boards in detail.

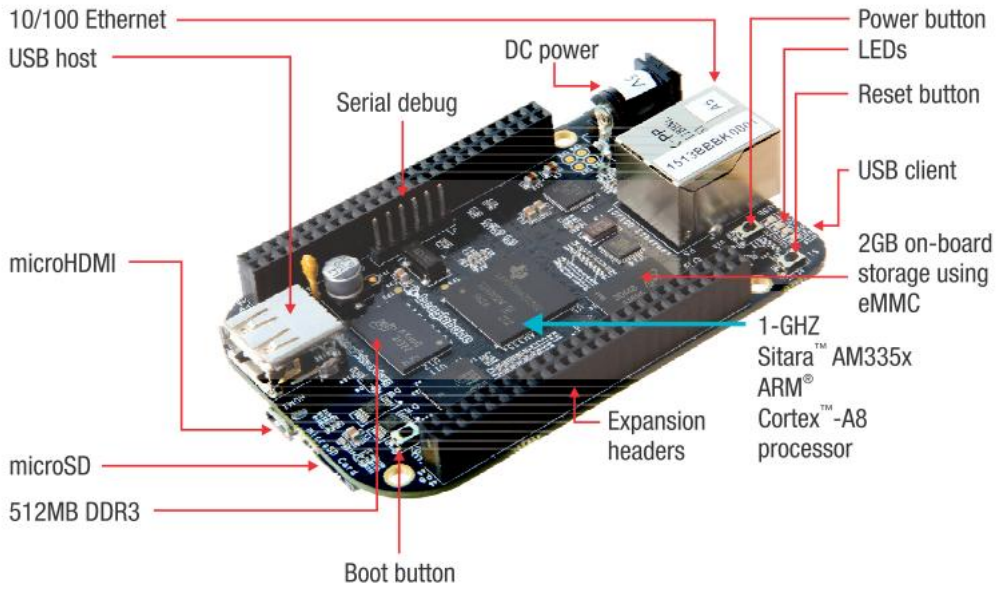


Figure 32: BeagleBone Black PCB connectors (Kridner, et al., 2013)

	BeagleBone Black	BeagleBone
Processor	Sitara AM3359AZCZ100 1GHz, 2000 MIPS	AM3359 500MHZ-USB Powered 720MHZ-DC Powered
Graphics Engine	SGX530 3D, 20M Polygons/S	-
SDRAM Memory	512MB DDR3L 800MHZ	256MB DDR2 400MHZ (128MB Optional)
Onboard Flash	2GB, 8bit Embedded MMC	-
PMIC	TPS65217C PMIC regulator and one additional LDO.	TPS65217B Power regulators LiION Single cell battery charger (via expansion) Reset 20mA LED Backlight driver, 39V, PWM (via expansion) (Additional components required)
Debug support	Optional Onboard 20-pin CTI JTAG Serial Header	USB to Serial Adapter miniUSB connector 4 USER LEDs On Board JTAG via USB
Power source	miniUSB USB DC Jack 5VDC External Via Expansion Header	USB or 5.0VDC to 5.2VDC
PCB	3.4" x 2.1" 6 layers	
Indicators	1-Power 4-User Controllable LEDs 2-Ethernet	Power 4-User Controllable LEDs
HS USB 2.0 Client Port	Access to USB0, Client mode via miniUSB	Access to the USB1 Client mode
HS USB 2.0 Host Port	Access to USB1, Type A Socket, 500mA LS/FS/HS	USB Type A Socket, 500mA LS/FS/HS
Serial Port	UART0 access via 6 pin 3.3V TTL Header. Header is populated	-
Ethernet	10/100, RJ45	
SD/MMC Connector	microSD , 3.3V	
User Interface	Reset Button Boot Button Power Button	Reset Button
Video Out	16b HDMI, 1280x1024 (MAX) 1024x768,1280x720,1440x900, 1920x1080@24Hz w/EDID Support	-
Audio	Via HDMI Interface, Stereo	-
Expansion Connectors	Power 5V, 3.3V , VDD_ADC(1.8V) 3.3V I/O on all signals McASP0, SPI1, I2C, GPIO(65), LCD, GPMC, MMC1, MMC2, 7 AIN(1.8V MAX), 4 Timers, 4 Serial Ports, CAN0, EHRPWM(0,2),XDMA Interrupt, Power button, Expansion Board ID (Up to 4 can be stacked)	Power 5V, 3.3V , VDD_ADC(1.8V) 3.3V I/O on all signals McASP0, SPI1, I2C, GPIO(65), LCD, GPMC, MMC1, MMC2, 7 AIN(1.8V MAX), 4 Timers, 3 Serial Ports, CAN0, EHRPWM(0,2),XDMA Interrupt, Power button, Battery Charger, LED Backlight, Expansion Board ID (Up to 3 can be stacked)
Weight	39.68 g	

Chart 9: BeagleBone Black and BeagleBone features (Coley, et al., 2013) (Coley, 2012)

2.3.2.3. BeagleBoard-xM

2.3.2.3.1. Introduction

BeagleBoard-xM board is the newer version of the original BeagleBoard, and they are laptop-like boards. The better performance of BeagleBoard-xM and its new added connector make this improved version more expensive.

Some projects in which BeagleBoard-xM is involved are BeagleSNES where BeagleBoard-xM is able to run Super Nintendo game titles; Robotic Camera Operator where it is used to track vehicles at over 30MPH; or LVDS LCD Add-on Board in which BeagleBoard-xM is connected to an additional PCB which provides interface to LVDS LCD panels.

2.3.2.3.2. Hardware blocks

This section provides an overview of the key components of the BeagleBoard-xM. However, the figure below shows the key components compared with its predecessor to observe the evolution of the board.

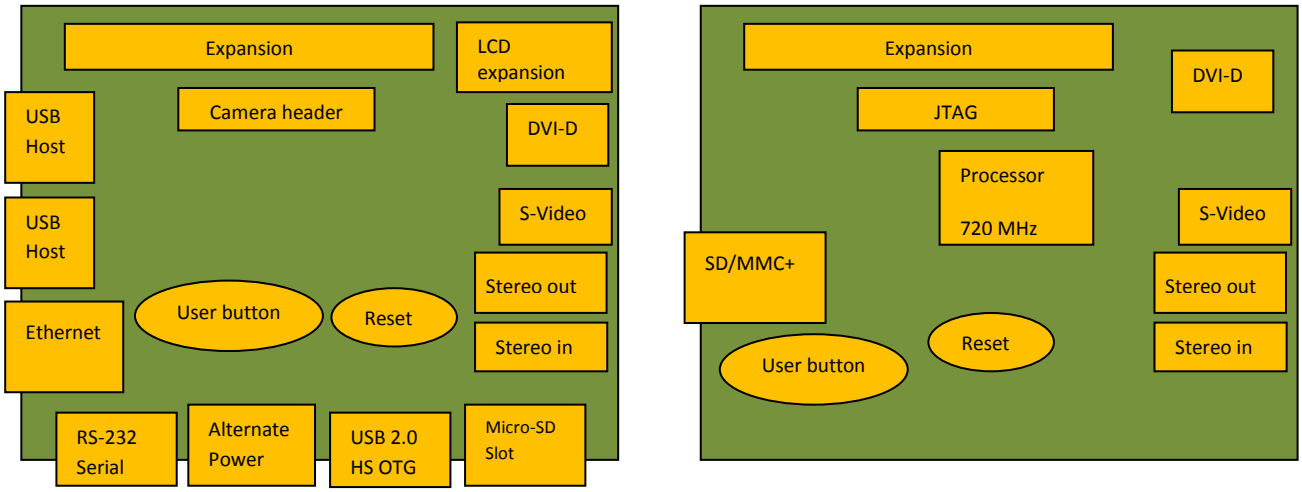


Figure 33: BeagleBoard-xM (left) and BeagleBoard hardware blocks

- **Processor** –the core component of the BeagleBoard-xM is the processor DM3730CBP 1 GHz. It is high-performance and multimedia application device, and its architecture is able to embed high-level operating systems like Windows CE, Linux, QNX, Symbian, etc. The operating systems recommended are Amgstrong Linux, Android, Ubuntu and XBMC.
- **Power supply** – can be provided via HS USB OTG port if USB Host ports are not used, otherwise a wall 5 V supply must be plugged to the board.
- **Indicators** – the board is equipped with five green LEDs programmable by the user; one on the TPS65950 – this component is responsible for managing power – controlled via I2C interface, two on the processor programmed via GPIO pins, one Power LED

which indicates if the board is turned on, and other LED to alert the user if the voltage range is exceeded.

- **Connectors** – S-Video connector that provides S-Video output of the board to a standard TV, and can be support the formats NTSC (by default) and PAL; a DVI-D connector which can be used to connect with a LCD monitor; a LCD header –to incorporate LCD panels to the board; RS232 DB9 Connector which can be connected a USB to serial cable; and a camera connector that supports VGA, 2MP, 3MP, and 5MP resolutions. Furthermore, the board integrates a main expansion header to allow the connection of custom circuitry and expand the functionality of the board.

2.3.2.3.3. Technical features

The figure above exposes the locations of the key components on the PCB layout of the BeagleBoard-xM board. In addition a table is exposed to compare the technical specifications of BeagleBoard and BeagleBoard-xM boards in detail.

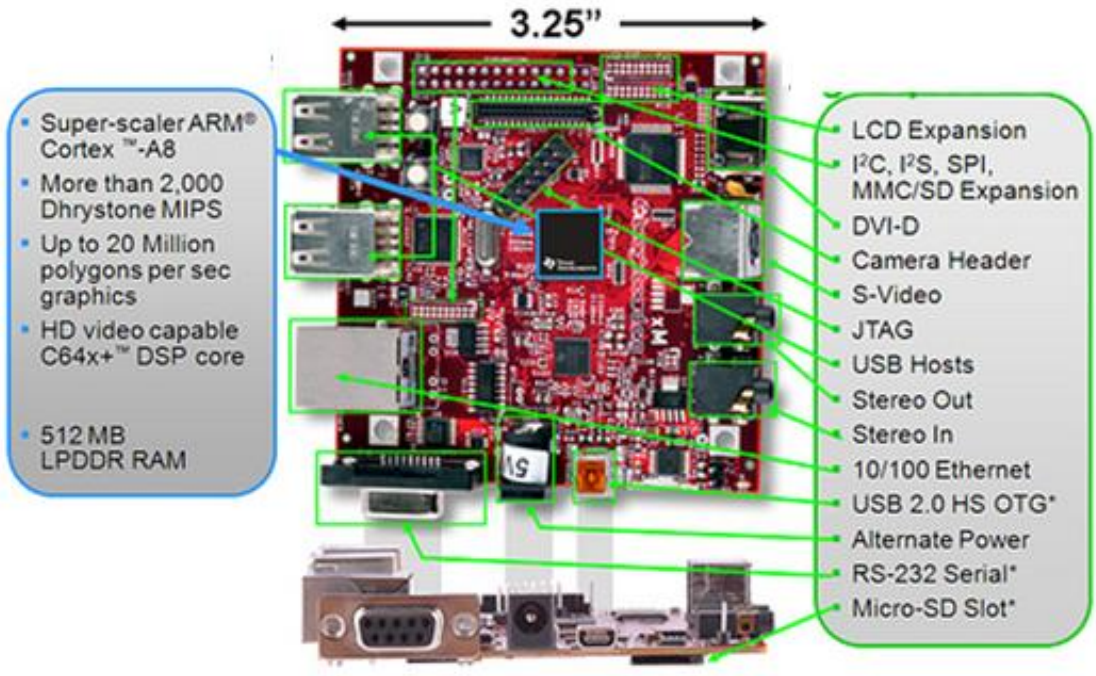


Figure 34: BeagleBoard-xM detailed pinout (RedSAM, 2013)

	BeagleBoard-xM	BeagleBoard
Processor	Texas Instruments Cortex A8 1GHz processor	OMAP3530DCBB72 720MHz
POP Memory	Micron 4Gb MDDR SDRAM (512MB) 200MHz	Micron 2 GB NAND (256 MB) 4 GB C5 2 GB MDDR SDRAM (256 MB)

PMIC TPS65950	Power regulators, Audio CODEC, Reset, USB OTG PHY	
Debug Support	14-pin JTAG	GPIO Pins
	UART	3 LEDs
PCB	3.1" x 3.0" (78.74 x 76.2mm) 6 layers	
Indicators	Power, Power Error PMU 2-User Controllable USB Power	Power, Power Error PMU 2-User Controllable
HS USB 2.0 OTG Port	Mini AB USB connector TPS65950 I/F	
HS USB Host Port	SMSC LAN9514 Ethernet HUB 4 FS/LS/HS Up to 500ma per Port if adequate power is supplied	Single USB HS Port Up to 500 Ma Power
Ethernet	10/100 From USB HUB	-
Audio connectors	L + R out 3.5 mm L + R Stereo In 3.5 mm	
SD/MMC Connector	MicroSD	6 in 1 SD/MMC/SDIO 4/8 bit support, Dual voltage
User Interface	1-User defined button Reset button	
Video	DVI-D S-Video	
Camera	Connector	Supports Leopard Imaging Module
Power supply	USB Power DC Power	
Overvoltage Protection	Shutdown @ Over voltage	-
Main Expansion Connector	Power (5V & 1.8V) McBSP I2C MMC2 UART McSPI GPIO PWM	Power (5V & 1.8V) McBSP I2C MMC UART McSPI GPIO PWM
2 LCD Connectors	Access to all of the LCD control signals plus I2C 3.3V, 5 V, 1.8 V	
Auxiliary Audio	4 pin connector McBSP2	-
Auxiliary Expansion	MMC3 GPIO, ADC, HDQ	-

Chart 10: BeagleBoard-xM technical features (Coley_b, 2012) (Coley_c, 2012)

CHAPTER 3: CLOUD PLATFORMS

3.1. Introduction

In this chapter, I introduce five cloud platforms specially developed for the Internet of Things.

Web platforms are built upon cloud computing paradigm which integrates massive scale distributed systems enabled by multiple evolving technologies related to resource virtualization. They are featured by ubiquitous access and shared data, and their key factor is abstraction; data is stored in locations that are unknown and applications run on physical systems that are not specified.

Platforms provide an interface as an access point to their resources, accessible over internet protocols. They use abstraction mechanisms for mapping a logical address to a physical resource in the form of Application Programming Interface (API) REST.

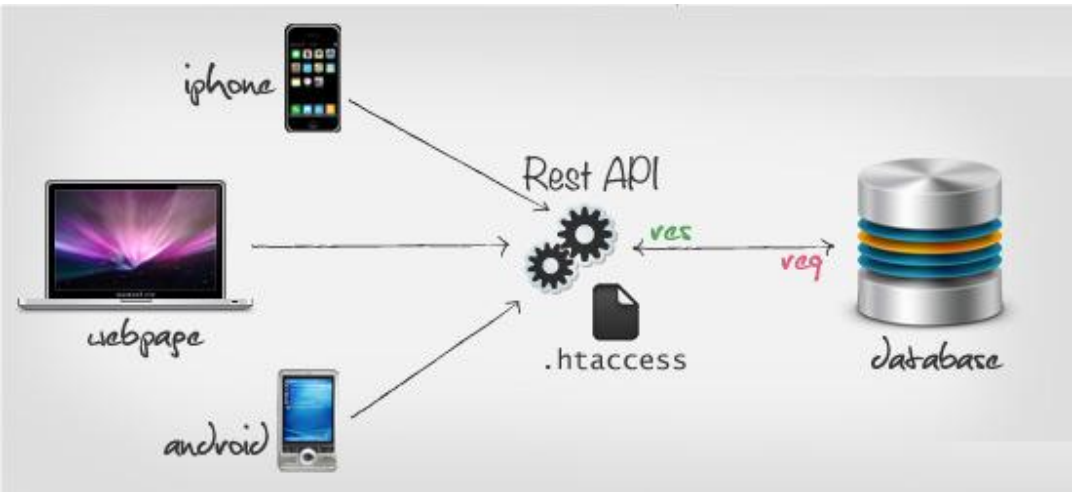


Figure 35: High overview of a web platform interconecction (SmartSell, 2013)

Representational State Transfer (REST) makes building distributed systems based on the notion of resources. Accessing information resources is made by identifying each of them with a global identifier assigned by REST. Identifiers are URIs expressed in HTTP way. Therefore, using HTTP commands (requests) various network clients can communicate with resources. With REST, clients can exchange information with servers in the form of documents or files. There are a variety of data formats for exchanging data such as text, an image file, XML, JSON, CSV, etc. Most of platforms use JSON by default and provide XML or CSV as second options.

- XML (Extensible Markup Language) is particularly useful for interacting with existing systems such as building management systems. However it is not recommended for constrained devices because it requires a more complex parser to extract data and building messages to transmit.

- JSON (JavaScript Object Notation) format is the preferred for web based applications. Compared to XML, JSON uses less bandwidth to transmit, and has much lower processing overheads, therefore it can be easily parsed.
- CSV (Comma Separated Values) is the simplest data format and is well suited for use by embedded devices. The simplicity of its structure makes it to differ from JSON and XML; while these two formats can send the same information and its hierarchy, CSV carries values separated by commas, therefore it is limited to represent simple data hierarchy.

In order to appreciate the differences among the mentioned data formats, three examples are exposed in the figures below. Note that the information represented is the same, and the "lightest" representation corresponds with CSV, while the "heaviest" with XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<eeml>
  <environment>
      <data id="example">
      <value>123</value>
    </data>
      <data id="key">
      <value>value</value>
    </data>
      <data id="datastream">
      <value>456</value>
    </data>
  </environment>
</eeml>
```

```
{
  "version":"1.0.0",
  "datastreams" :
  [
    {
          "id" : "example",
        "value" : "123"
    },
    {
          "id" : "key",
          "value" : "value"
    },
    {
      "id" : "datastream",
      "value" : "456"
    }
  ]
}
```

Figure 36: Examples of data representation in XML format (left) and JSON (right)

```
example, 123
key, value
datastream, 456
```

Figure 37: Example of data representation in CSV format

3.2. Open.Sen.se

3.2.1. Introduction

Open.Sen.se (Open.Sen.se, 2013) is a web platform for testing and prototyping all kind of scenarios associated with the Internet of Things such as new devices and applications. Its intended audiences are designers, developers, artists and casual users. The platform was launched on November 2010 and is available in beta version, so it requires an invitation to create a user account.

sense *"Feel. Act. Make sense."*

Figure 38: Sen.se logo (Open.Sen.se, 2013)

User account offers a private dashboard with 23 mashup tools available (up to date) for handling flows of data captured by devices, for example; "eMail it" to automatically send data collected to a personalized e-mail; "Data Funnel" that combines and process the data from several sources; "Tweet it" which posts data in a Twitter account; "Post to Facebook" the same as "Tweet it" App; and other tools to display the data and process mathematic calculations with numeric data. The following figure shows the dashboard demo.

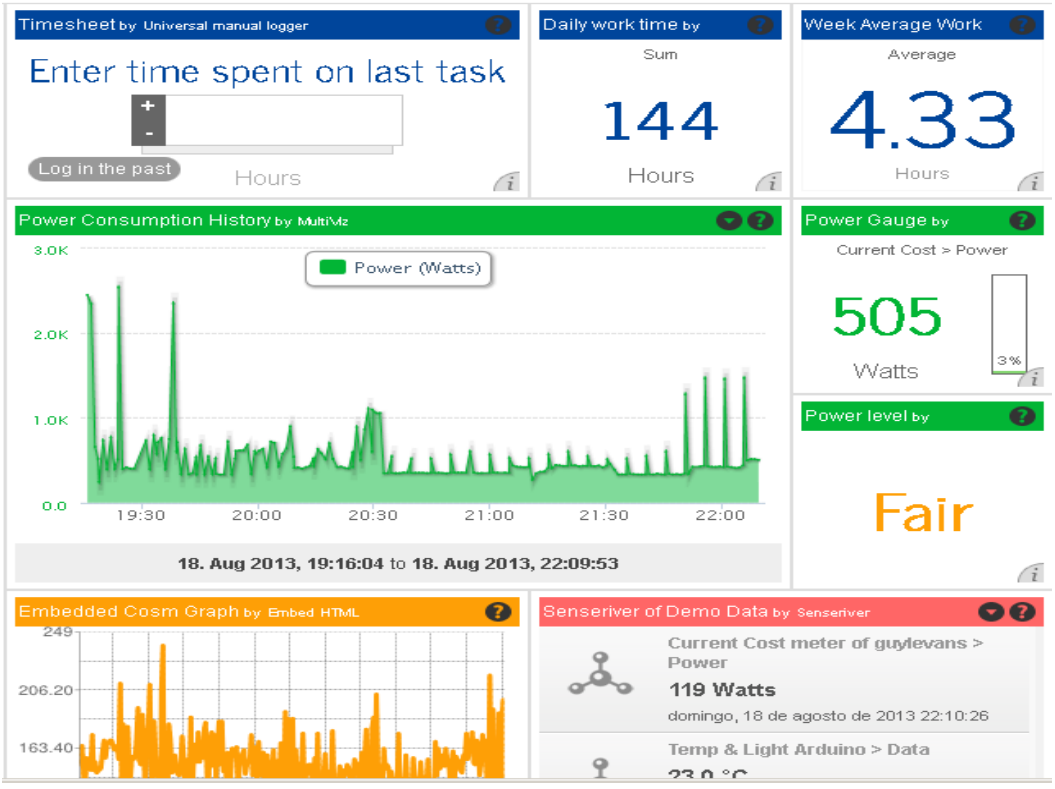


Figure 39: Dashboard demo of Open.Sen.se platform (Open.Sen.se, 2013)

3.2.2. Sen.se API

Open.Sen.se platform (hereafter the Platform) allows developers (hereafter Users) connecting any physical object (hereafter Devices) with TCP socket interface for connecting to the Internet through WiFi, Ethernet, etc. Sen.se API is responsible for handling the access to the platform. It is based on REST and permits to exchange data through HTTP requests with authentication (a key per user account).

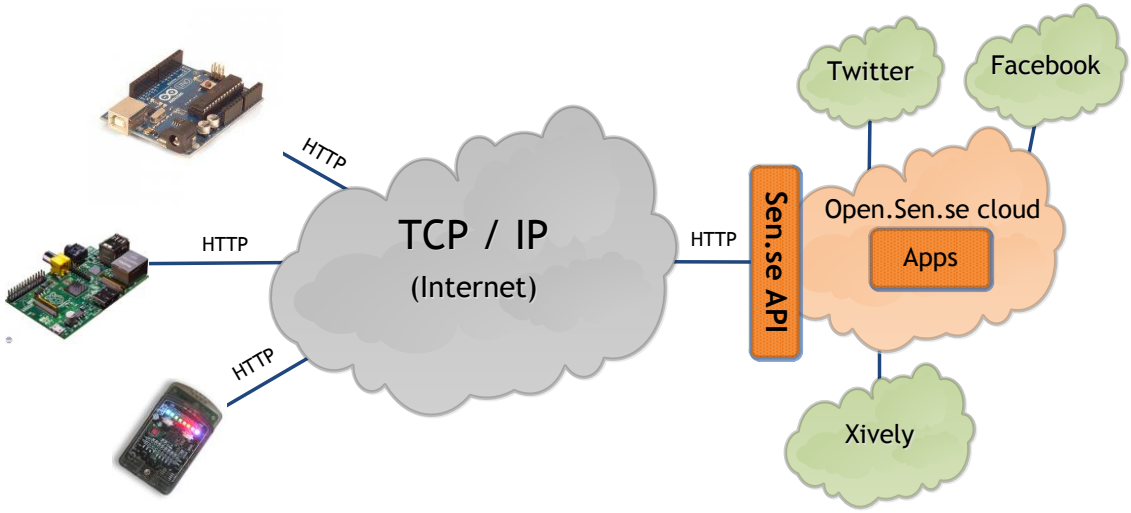


Figure 40: Overview of Open.Sen.se platform

Devices are registered in the Platform with information such as name, description, etc., and the API identifies them by their "Device Id". Devices will call the API for sending HTTP requests with the data they capture like state, activity and information about their environment. Both HTTP requests and responses support JSON format.

HTTP responses' fields depend on the requests made to the API. The API provides a set of HTTP responses in order to report clients if their requests were processed successfully or not; 200 (OK) if the request had succeeded; 400 (Bad request) if the request could not be understood due to malformed syntax; 404 (Not found) if the request-URI does not match with anything; and 500 (Internal Server Error) if the server could not process the request when this one is properly formed. The following figures show examples of requests and responses in JSON format and a table with HTTP parameters needed for each type of operation.

```
POST /events HTTP/1.1
Host: api.sen.se
Connection: close
sense_key = "SEN.SE_KEY"
Content-Type: application/json
Content-Length: (number of characters in message)

{
  "feed_id": 18400,
  "value": 25
  "timetag":"2013-01-28T09:11:00.000+00:00"  [Opcional field]
}
```

Figure 41: Example of HTTP request for publishing Events in Open.Sen.se platform

Action performed	Method	Endpoint
Publishing Events	POST	/events/
Get Feed last Event	GET	/feeds/_feed_id_/last_event/
Get Feed Events	GET	/feeds/_feed_id_/events/
Get Device last Event	GET	/devices/_device_id_/last_event/
Get Device events	GET	/devices/_device_id_/events/

Chart 11: HTTP parameters for calling the Sen.se API

```
[
   {
      "feed_id": __feed_id__,
      "value": __event_value__
   },
   {
      "feed_id": __another_feed_id__,
      "value": __another_event_value__
   }
]
```

```
{
 "feed_id": 18400,
 "value": "25",
 "publish_id": "004cafcdcc0b052",
 "id": "004cafcdcc0b052",
 "timetag": "2012-10-01T10:28:11.027+00:00"
}
```

Figure 42: Publishing request (left) and response (right) several Events in JSON.

Open.Sen.se call “Events” to the data captured that devices send to the Platform. Sending Events is called “publishing Events”, and published Events that come from the same sensor are stored in a “Feed” that can be used by other Devices or applications to trigger actions. Sen.se establishes a general “hierarchy” of their entities and their roles. The User is a person who owns an account on Sen.se; this User can register several Devices to her / his account. Each Device can have several feeds, input feeds associated to input data from Devices to the Platform (sensors) and output feeds associated to output data from the Platform towards Devices (actuators); and Events are the input data stored in input / output Feeds. The figure below shows the hierarchy.

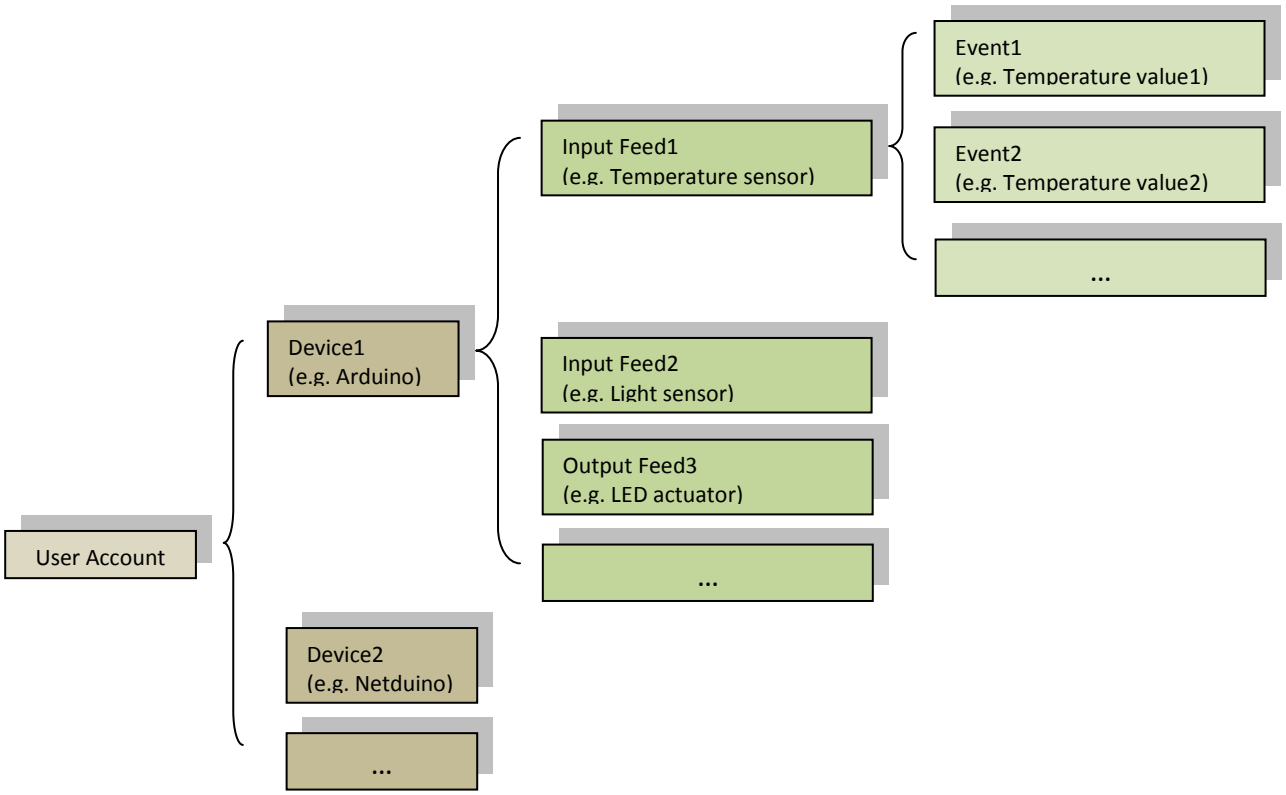


Figure 43: Open.Sen.se hierarchy of entities

3.3. ThingSpeak

3.3.1. Introduction

ThingSpeak (ThingSpeak, 2013) is a web platform powered by ioBridge, Inc., (ioBridge, 2013) a company that provides Web-enabled things solutions for manufacturers, professionals and casual users. ThingSpeak offers services for collecting data in real-time from devices, data processing, data visualizations for a wide range of applications such as sensor monitoring, energy monitoring, geo location tracking or interfacing with social networks. It provides an open-source API for connecting to the ThingSpeak's infrastructure which is supported by more than 500 servers.

ThingSpeak

Figure 44: ThingSpeak logo (ThingSpeak, 2013)

"An open platform designed to enable meaningful connections between things and people"

The following figure illustrates one of the Public Channels accessible from around the world.

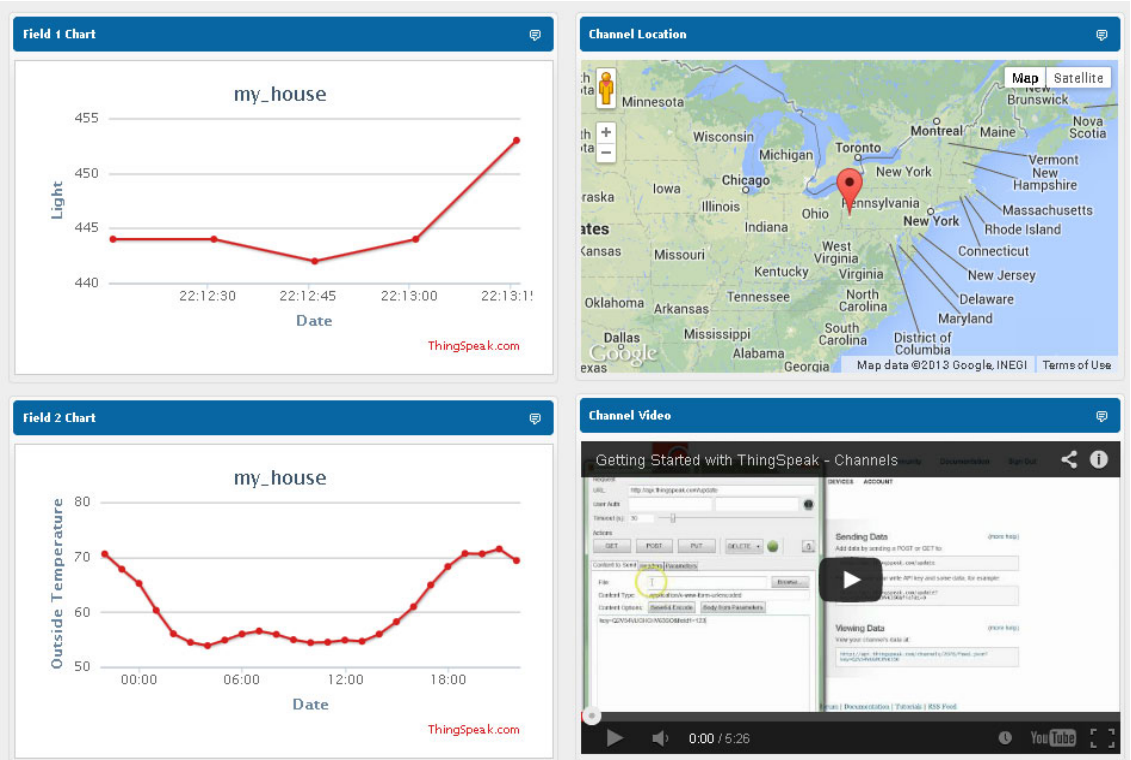


Figure 45: Public Channel of ThingSpeak platform (ThingSpeake, 2013)

3.3.2. ThingSpeak API

ThingSpeak API provides an interface for exchanging information among physical devices, ThingSpeak platform and third party entities such as Twitter, Prowl and Twilio. ThingSpeak platform allows any physical device with TCP socket can send HTTP requests to the API for storing any type of data or retrieving data.

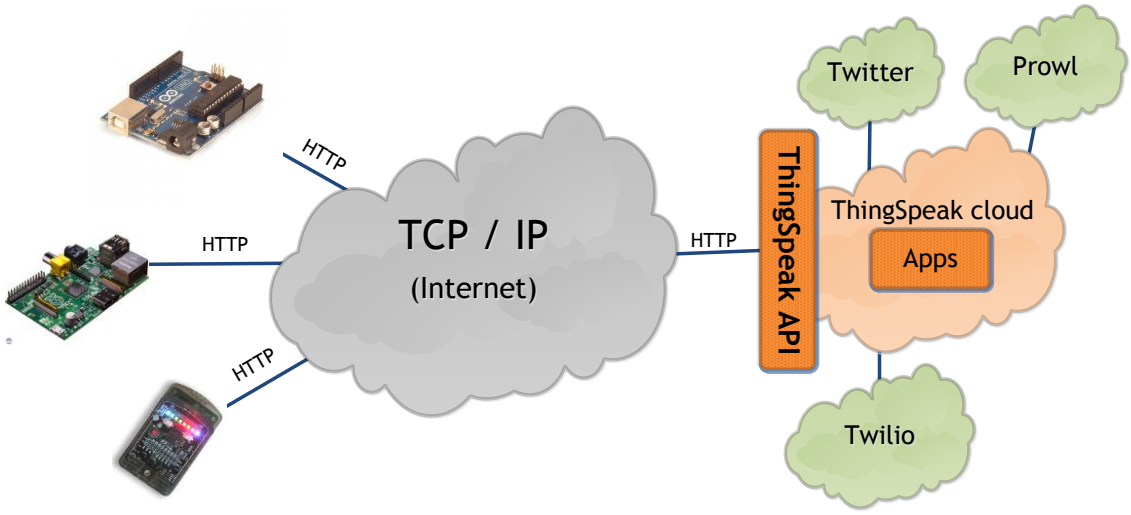


Figure 46: ThingSpeak platform

ThingSpeak platform stores data in "ThingSpeak Channels". Channels are represented as a web interface where stored data is published. They can be configured to be public for other people to view them or private, only accessible by signing into ThingSpeak.com user account.

Client applications are embedded in physical devices that can read and write to a ThingSpeak Channel using HTTP requests to ThingSpeak API. A channel classifies data in location information, status update and eight fields of data in which can be stored both numeric and alphanumeric formats. Each entry or feed is labeled with a unique Entry ID and is stored with a date and time stamp. Writing in a Channel requires a Write key to ensure that only authorized applications can access to their data. Numeric data can be processed such as time scaling, averaging, median, summing, and rounding. Channel feeds supports JSON, XML, and CSV formats.

Action performed	Method	Endpoint
Updating a channel	POST	/update
Retrieving channel feeds	GET	/channels/(channel_id)/feed.(format)
Retrieving the last entry in channel feed	GET	/channels/(channel_id)/feed/last.(format)
Retrieving a field feed	GET	/channels/(channel_id)/field/(field_id).(format)
Retrieving the last entry in a Field Feed	GET	/channels/(channel_id)/field/(field_id)/last.(format)
Retrieving status updates	GET	/channels/(channel_id)/status.(format)

Chart 12: Actions performed to ThingSpeak API

Chart API

It is used to display stored data of ThingSpeak channels in graphs. These graphs are called "charts" and can present numerical data and can be embedded in external websites. Charts can be static or dynamic; dynamic charts visualize data variation in real time.

ThingTweet

Twitter gives to his users the choice of updating their account through an Open Authentication. However, this option is not designed for devices with constrained processing capabilities such as Arduino or Netduino; ThingSpeak API solves the problem with ThingTweet APP. It roles a Twitter Proxy which sends status updates to Twitter through ThingSpeak API calls.

```
POST /apps/thingtweet/1/statuses/update HTTP/1.1
Host: api.thingspeak.com
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: (number of characters in message)
```

Figure 47: Updating a Twitter account through TweetThing App (ThingSpeakb, 2013)

ThingHTTP

ThingHTTP allows connectivity of constrained devices to any web services such as Prowl and Twilio using HTTP over a network or Internet. This App eliminates the necessity of the devices to implement the protocol to deal with such web services. In addition, it is possible to incorporate a "Parse String" within the HTTP request to that service to avoid of coding a parser in constrained devices.

```
POST /apps/thinghttp/send_request HTTP/1.1
Host: api.thingspeak.com
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: (number of characters in message)
```

Figure 48: Connecting with a third party entity through TweetHTTP App (ThingSpeakc, 2013)

TweetControl

Twitter Streaming API allows monitoring streams in real time. Doing this requires a dedicated server with long running process which poll Twitter for status updates. ThingControl App makes the mentioned task. Then the developer can focus on determine the stream and the specific "hasttag" and perform a control action such as sending a ThingHTTP request to third party entities.

React

ThingSpeak platform process streams of data and lets the possibility to set trigger actions when a condition is met regarding these data of a ThingSpeak Channel. The condition types depend on the specific data to monitor; sensor data, texts, etc. React manages String conditions, status, numeric, geographic location and for testing if the device stops sending data to the platform. When a condition is met React can trigger a ThingHTTP request to a third party entity or post a tweet with ThingTweet App.

Next figures shows an example of HTTP request for updating a ThingSpeak Channel and another example of ThingSpeak feed in JSON format.

```
POST /update HTTP/1.1
Host: api.thingspeak.com
Connection: close
X-THINGSPEAKAPIKEY: (Write API Key)
Content-Type: application/x-www-form-urlencoded
Content-Length: (number of characters in message)

field1=(Field 1 Data)&field2=(Field 2 Data)&field3=(Field 3 Data)&field4=(Field 4 Data)&field5=(Field 5 Data)&field6=(Field 6 Data)&field7=(Field 7 Data)&field8=(Field 8 Data)&lat=(Latitude in Decimal Degrees)&long=(Longitude in Decimal Degrees)&elevation=(Elevation in meters)&status=(140 Character Message)
```

Figure 49: HTTP request for updating a ThingSpeak Channel (ThingSpeakd, 2013)


```
{
"channel":
 {"created_at":"2010-12-14T01:20:06Z",
  "description":"Netduino Plus connected to sensors around the house",
  "field1":"Light",
  "field2":"Outside temperature",
  "id":9,
  "last_entry_id":4987701,
  "latitude":"40.44",
  "longitude":"-79.996",
  "name":"my_house",
  "updated_at":"2013-08-11T13:17:57Z"},
  "feeds":
  [
            {
                        "created_at":"2013-08-10T13:18:14Z",
                        "entry_id":4982339,
                        "field1":"386",
                        "field2":"56.050955414012741"
            },
            {
                        "created_at":"2013-08-10T13:18:29Z",
                        "entry_id":4982340,
                        "field1":"369",
                        "field2":"57.834394904458598"
            }
  ]
}
```


Figure 50: Example of ThingSpeak feed in JSON format (ThingSpeakd, 2013)

3.4. EVRYTHNG

3.4.1. Introduction

EVRYTHNG (EVRYTHNG, 2013) is an online platform for storing and sharing dynamic data of millions of real world physical objects. The core technology of EVRYTHNG is its engine, which handle issues such as provide an Active Digital Identity (ADI) for any physical object. An ADI is a web resource that represents the data associated to any physical object in the world. The EVRYTHNG API is responsible for accessing to the information attached to each ADI. Based on the stored data EVRYTHNG provides a developer program for building applications and services.

Make products smart. We make products smart, simple and social by connecting them to the Web.

Figure 51: EVRYTHNG logo (EVRYTHNG, 2013)

The next figure shows the dashboard of the developer program.

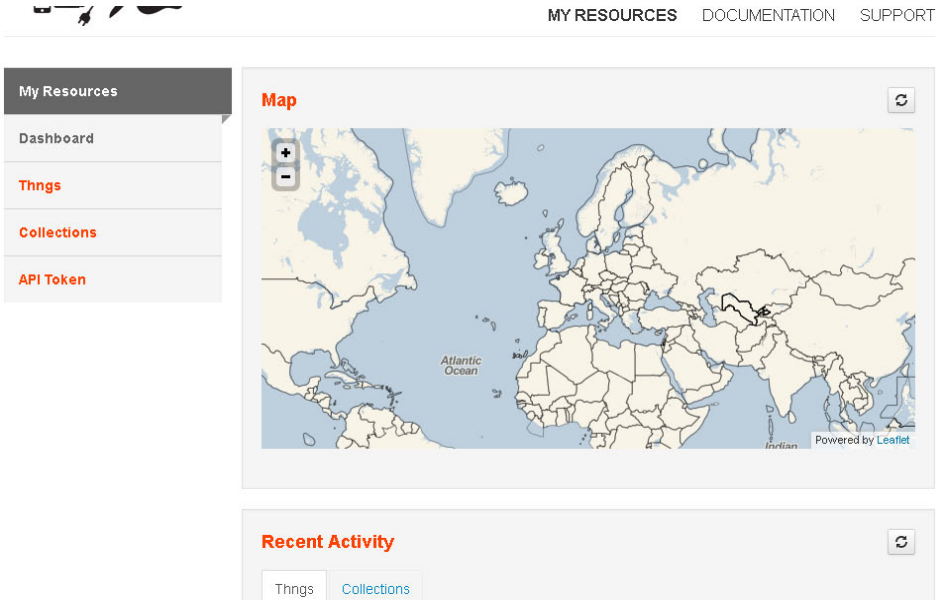


Figure 52: EVRYTHNG dashboard (EVRYTHNG_b, 2013)

3.4.2. EVRYTHING data hierarchy

"Thngs" are the central data structure in the EVRYTHNG engine. They contain all data generated by any physical object. For EVERYTHNG, a "Thng" is not a synonymous of physical object, but is also their associated data such as the geographic position and other specific information (depending on device). It also calls it Active Digital Entities (ADIs) because from a Web perspective they can be seen as resources which model the real-world.

EVRYTHNG allows “users” grouping thngs in “collections”, and it permits one thng can belong to several collections. All types of data which thngs provide to the platform are called “Properties” (e.g. temperature, humidity) that can be updated and retrieved at any time.

The following figure shows the hierarchy of mentioned entities.

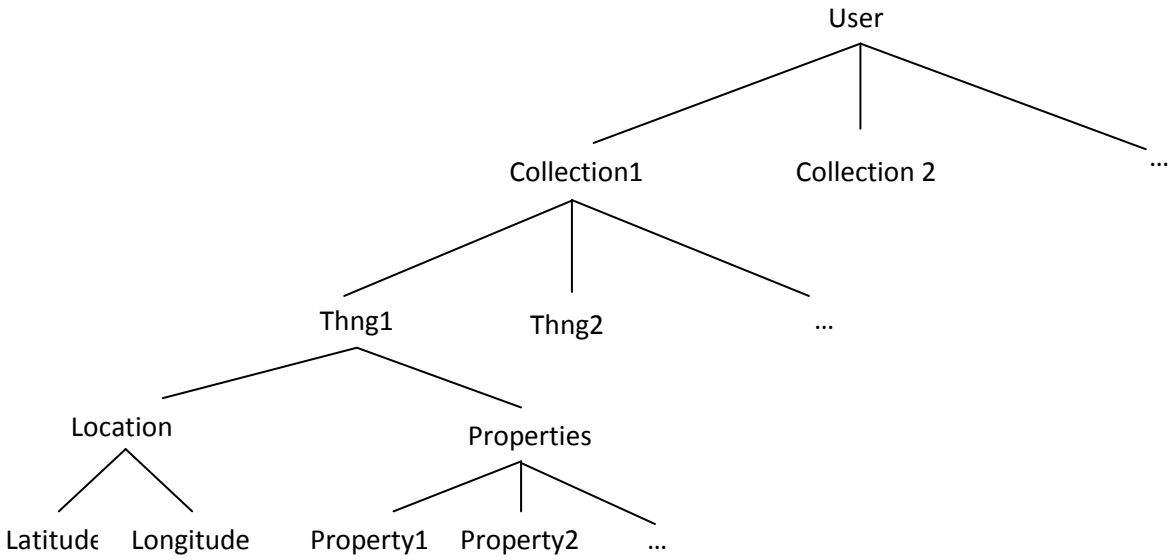


Figure 53: EVRYTHNG hierarchy data model

Both Thngs and Collections are labeled with a unique identifier by the API at creation time for accessing / editing them in the future.

3.4.3. EVRYTHNG API

When dealing with the EVERYTHNG back-end, HTTP requests and responses are required. The API of the EVRYTHNG engine follows the RESTful architectural principles. REST provides a means of constructing distributed systems based on the notion of “resources”; retrieving resources is associated to GET method, creating resources to POST, updating resources to PUT, and deleting them to DELETE method. EVERYTHNG consider Thngs, Collections, Properties, and Locations as resources. As a consequence, EVERYTHNG API will provide the properly methods for the each action performed.

On the other hand, every request to the API must include an API key using Authorization HTTP header to identify the user or application issuing the request and execute it if authorized. The API accepts both HTTPS and insecure HTTP; EVRYTHNG recommends using HTTPS, HTTP is for low-power devices without SSL support.

The API will reply requests with any of the following codes. Regarding HTTP code errors, the API replies with a JSON payload with the error message associated to help HTTP client handling them.

HTTP request	Description
200 OK	When requests for updating or retrieving actions were successfully processed
201 Created	When requests for creating a new resource were succeeded.
400 Bad Request	This reply would be due to a malformed JSON document.
401 Unauthorized	The requester (HTTP client) does not include the authentication field or this is invalid.
403 Forbidden	The authentication field is valid but there is no access to the resource.
404 Not found	The API does not match the requested resource with any of the available ones.
415 Unsupported Media Type	The API refuses requests with no JSON format.
500 Internal Server Error	The server could not fulfill the request due to an internal code.
503 Service Unavailable	It is used when EVRYTHNG servers are overloaded or when external services they relay did not respond.

Chart 13: HTTP status codes of EVRYTHNG API

The following charts specify in detail the HTTP requests parameters for managing the resources of the platform: thngs, their properties, their locations, and collections of them. Sending successful requests require at least one thng created, so the first step is "creating a single thng".

Thngs

Action performed	Method	Endpoint
Create a single thng	POST	/thngs
Get a single thng	GET	/thngs/{thngID}
Update a thng	PUT	/thngs/{thngID}
Deleting a thng	DELETE	/thngs/{thngID}
Get the list of all thngs	GET	/thngs

Chart 14: Actions performed to Thngs of EVRYTHNG API

Properties

Get a single property	GET	/thngs/{ThngID}/properties/{key}[?from=<timestamp>&to=<timestamp>]
Update a single property	PUT	/thngs/{ThngID}/properties/{key}
Updating properties	PUT	/thngs/{thngID}/properties
Deleting individual properties	DELETE	/thngs/{ThngID}/properties/{key}[?to=<timestamp>]
Get the list of all properties	GET	/thngs/{thngID}/properties

Chart 15: Actions performed to thngs' properties of EVRYTHNG API

Locations

Read the locations	GET	/thngs/{ThngID}/location
Update the location	PUT	/thngs/{ThngID}/location
Deleting location updates	DELETE	/thngs/{ThngID}/location[?to=<timestamp>]

Chart 16: Actions performed to locations of thngs of EVRYTHNG API

Collections

Creating a collection of thngs	POST	/collections
Get a single collection	GET	/collections/{CollectionID}
Editing a collection	PUT	/collections/{CollectionID}
Adding (existing) thngs to a collection	PUT	/collections/{CollectionID}/thngs
Deleting a collection	DELETE	/collections/{CollectionID}
Removing a thng from a collection	DELETE	/collections/{CollectionID}/thngs/{ThngID}
Removing all thngs from a collection	DELETE	/collections/{CollectionID}/thngs/
List all thngs in a collection	GET	/collections/{CollectionID}/thngs
Get all collections	GET	/collections/

Chart 17: Actions performed to collections of thngs of EVRYTHNG API

The JSON document for describing a thng is as follows:

```
<Thng>={
  "id": <String>,
  "createdAt": <timestamp>,
  "updatedAt": <timestamp>,
  "name": <String>,
  "description": <String>,
  "location": {
  "longitude": <Number>,
    "latitude": <Number>,
    "timestamp": <timestamp>
  },
  "product": <String>,
  "properties": {
    <String>: <String>,
      ... },
  "tags": [<String>, ...]
}
```

Figure 54: JSON document for defining a "thng" of EVRYTHNG API (EVRYTHNG_c, 2013)

The figures below show the HTTP request for creating a single thing and its respective HTTP response. Note that the response will provide the unique identifier of the thng, (thngID=504766a7c8f205a0744243c1), which can be only modified by the EVRYTHNG engine. When creating a thng, the “name” field is the only one mandatory.

```
POST /thngs
Content-Type:
application/json
Authorization:
$EVRYTHNG_API_KEY
{
  *"name": <String>,
  "description": <String>,
  "location": {
    *"longitude": <Number>,
    *"latitude": <Number>,
    "timestamp": <timestamp>
  },
  "product": <String>,
  "properties": {
    <String>: <String>,
      ...},
  "tags": [<String>, ...]
}
```

```
HTTP/1.1 201 Created
Content-Type: application/json
Location:
https://api.evrythng.com/thngs/504
766a7c8f205a0744243c1
{
  "id":
"504766a7c8f205a0744243c1",
  "createdAt": 1346856615704,
  "updatedAt": 1346856615704,
  "name": "Room-212-Santa-Maria-
Novella",
  "description": "Room 212 at the
Santa-Maria Novella in Florence,
Italy",
  "location": {
    "latitude": 43.772828,
    "longitude": 11.249488},
  "properties":{
    "pricePerNightEuros":"300",
    "pricePerNightGBP":"200",
    "RoomSize":"40sqm",
    "Style":"Double Bed, 2pax"},

"tags":["room","hotel","Florence",
"romance"]
}
```

Figure 55: Request (left) and response (right) for creating a new thng (EVRYTHNG_c, 2013)

3.5. Xively

3.5.1. Introduction

Xively (Xively, 2103) claims to be the first cloud for the Internet of Things in the world; it manages data from 250 million devices and it has 17 million users. It was the old Pachube which was acquired by LogMeIn, Inc. in July 2011.

The Internet of Things is Open for Business

Figure 56: Xively logo (Xively, 2103)

Xively provides a platform for data storage in real time, selective data sharing, and device management services, among others. Any user of the world can access for free to Xively Developer Workbench and Xively Management Console for connecting devices and programming web applications with the Xively API. It has many tested devices such as Arduino or Raspberry Pi.

3.5.2. Xively data hierarchy

Xively organizes their data in different layers. The bottom layer is associated to the data or metadata captured by devices and uploaded to the platform. Higher layers encapsulate and organize data in order to build applications and make products.

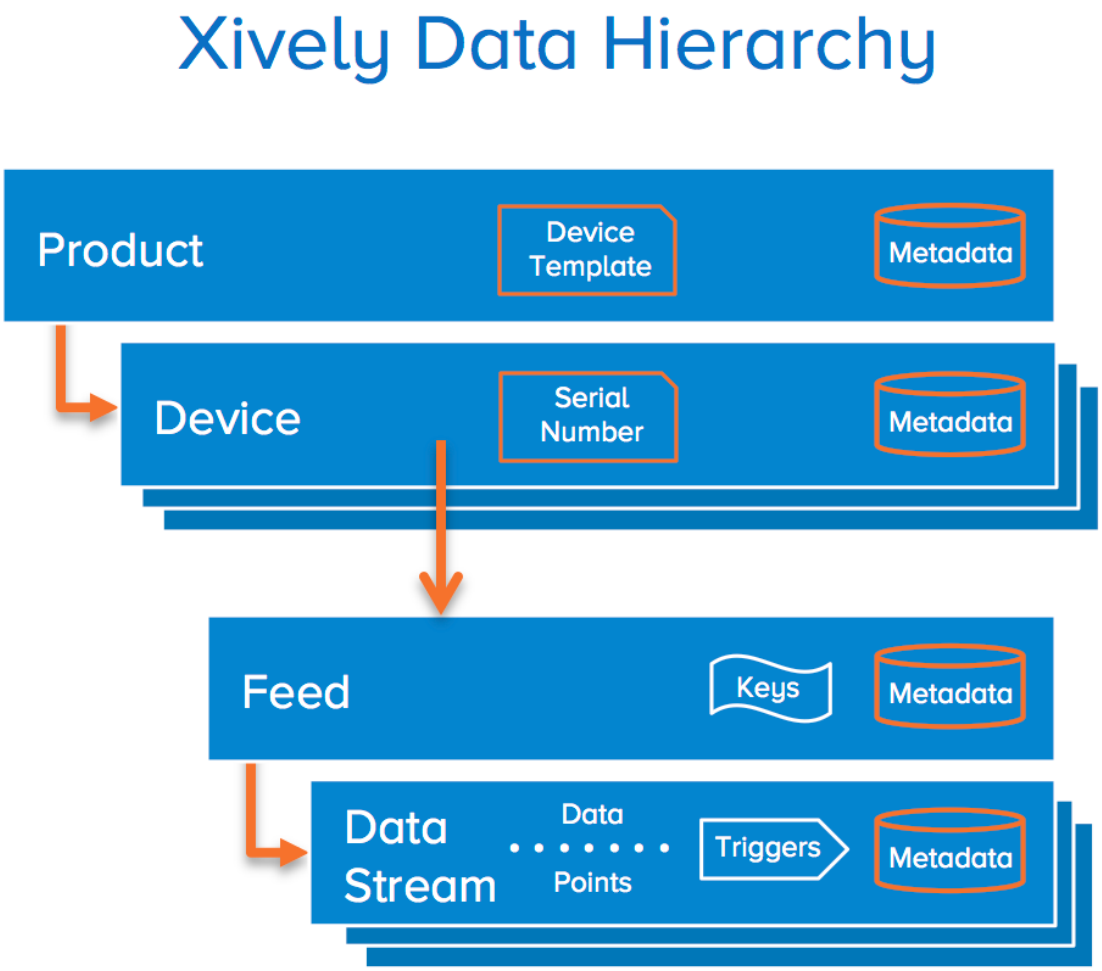


Figure 57: Xively data hierarchy (Xively_b, 2013)

The highest level of the hierarchy corresponds to Products. They are abstract entities which encompass devices, and represent the applications and services built upon the data provided by devices. One product can combine services and applications of several devices. Product definition is specified with a product name, a description and other optional parameters.

The second level is reserved for devices which are individual physical objects which provide and retrieve data to and from the platform. They are identified by a serial number that can consist of any combination of alphanumeric characters. Any device is matched with one "Feed" which represents all types of data that a device is able to manage (e.g. an Arduino device can have a temperature sensor and a light sensor). Feeds can be public or private; public Feeds mean that the device and its data are available to any user under the terms of the Creative Commons

CCO 1.0 Universal license; by contrast, private Feeds refer that device's data are available only to the developer or selective people.

These types of data are called "Datastreams" which are uniquely identified, and represent the data flows that go towards the platform and viceversa (e.g. a Netduino device that sends continuously temperature values). Xively calls "Datapoint" to each data value archived in a discrete time of a Datastream.

3.5.3. Xively API

Xively API acts an interface for reading and writing data to the Xively Cloud Service and is based on REST principles. Most HTTP requests support JSON, XML and CSV format (JSON is the format by default). Xively API can be securely called over IPv4 or IPv6 via HTTP, HTTPS (is the protocol by default), sockets or websockets and MQTT protocols. Every request require authentication via an API key. In addition, it supports OAuth, so third-party entities can communicate with the API.

The Xively API will return appropriate HTTP status codes for every request.

HTTP request	Description
200 OK	Request were successfully processed
400 Bad Request	This reply would be due to a malformed JSON document.
401 Unauthorized	The requester (HTTP client) does not include the authentication field or this is invalid.
403 Forbidden	The authentication field is valid but there is no access to the resource.
404 Not found	The API does not match the requested resource with any of the available ones.
406 Not acceptable	It due to the wrong document type
422 Unprocessable entity	Xively could not create a Feed because the payload was malformed.
500 Internal Server Error	The server could not fulfill the request due to an internal code.
503 No server error	It is used when Xively servers are overloaded

Chart 18: HTTP status codes of Xively API

Xively API manages 7 resources for performing operations to interact with the platform: Products, Devices, Keys, Feeds, Triggers, Datastreams, and Datapoints. Operations can be classified in storing, reading and writing:

- Stored data can be deleted in the following ways: individual Datapoints, multiple Datapoints by specifying a range, and entire Datastream.
- Reading can be made via three resources: Feeds, Datastreams and Datapoints. Xively API allows reading a single Feed, single Datastream, range of historical Datapoints, and all Feeds.
- Writing Datapoints to Datastreams can be accomplished by devices, applications and services in the following ways: single Datapoint to a single Datastream, single

Datapoint to multiple Datastreams, multiple Datapoints to a single Datastream and multiple Datapoints to multiple Datastreams.

The following tables list the actions which can be performed to each resource. All actions support JSON format by default, however, some of them are able to reply body responses in alternative formats such as XML, CSV and PNG.

Feeds

Action performed	Method	Endpoint
Read	GET	/feeds/
Read a range	GET	/feeds/feed_id?range
Update	PUT	/feeds/feed_id
Lista ll	GET	/feeds

Chart 19: Actions performed to Feeds in Xively API

Requests of the Feeds' table admit JSON, XML and CSV format.

Datastreams

Action performed	Method	Endpoint
Create	POST	/feeds/feed_id/
Read	GET	/feeds/feed_id/
Read a range	GET	/feeds/feed_id/datastreams/datastream_id?range
Update	PUT	/feeds/feed_id/datastreams/datastream_id
Delete	DELETE	/feeds/feed_id/datastreams/datastream_id
List all	GET	/feeds/feed_id

Chart 20: Actions performed to Datastreams in Xively API

Requests of the Datastreams' table of GET method support JSON, XML, CSV and PNG. The rest does not support PNG.

Datapoints

Action performed	Method	Endpoint
Delete (single)	DELETE	/feeds/feed_id/datastreams/datastream_id/datapoints/timestamp
Delete (range)	DELETE	/feeds/feed_id/datastreams/datastream_id/datapoints?range

Chart 21: Actions performed to Datapoints in Xively API

Products

Action performed	Method	Endpoint
Create	POST	/products
Update	PUT	/products/product_id
List All	GET	/products
Read	GET	/products/product_id
Delete	DELETE	/products/product_id

Chart 22: Actions performed to Products in Xively API

Devices

Action performed	Method	Endpoint
Create	POST	/products/product_id/devices

Read	GET	/products/product_id/devices/serial_number
Update	PUT	/products/product_id/devices/serial_number
Delete	DELETE	/products/product_id/devices/serial_number
List All	GET	/products/product_id/devices
Activate	GET	/devices/activation_code/activate

Chart 23: Actions performed to Devices in Xively API

Requests of Devices' table only support JSON format except the action "Activate" which support JSON and CSV formats.

Keys

Action performed	Method	Endpoint
Create	POST	/keys
Read	GET	/keys/key_id
Delete	DELETE	/keys/key_id
List All	GET	/keys

Chart 24: Actions performed to Keys in Xively API

Requests of Keys' table support JSON and XML.

Triggers

Action performed	Method	Endpoint
Create	POST	/triggers/
Read	GET	/triggers/trigger_id
Update	PUT	/triggers/trigger_id
Delete	DELETE	/triggers/trigger_id
List All	GET	/triggers/

Chart 25: Actions performed to Triggers in Xively API

Requests of Triggers' table support JSON and XML. Triggers are used for sending notifications in the form of HTTP POST requests to a URL when a monitored Datastream satisfies a preconfigured condition.

The figures below show several examples of HTTP responses body in different formats. The following figures show examples of HTTP responses bodies. The shown messages do not require HTTP request body. Some of them are exposed in different data formats in order to appreciate the differences and compare with each other. Each action performed implies a HTTP request and response. The figures below show three responses; first "create a Product" in JSON format, second "activate a Device" in JSON and CSV formats, and third "read a single Datastream" in JSON, XML, CSV and PNG data formats.

1) "Create a Product" request body in JSON format.

Products, Feeds and Datastreams can add value to its data with optional tags, keywords that help them to be found, grouped and / or identified.

```
{"product":{
    "description":"this is a device on xively",
    "name":"API Example",
    "feed_defaults":{
      "title":"Demo Unit",
      "tags":["device:type=API Example"],
      "private":"true",
      "datastreams":[{"id":"channel1"},{"id":"demo"}]},
      "user":"USERNAME"
    }
}
```

Figure 58: HTTP body response for creating a Product (Xively_c, 2013)

2) "Activate a Device" Response body request in JSON and CSV formats.

```
{
    "apikey": "API_KEY_HERE",
    "feed_id": FEED_ID_HERE,
    "datastreams": [ "temperature"
]
```

```
API_KEY_HERE,FEED_ID_H
ERE,NUMBER_OF_CHANNELS
_HERE,CHANNEL_NAME_HER
E
```

Figure 59: Activating a Device in JSON format (left) and CSV (right) (Xively_d, 2013)

3) "Read a single Datastream" body request in JSON, XML, CSV, and PNG, respectively.

The XML body is the "heaviest", therefore the client will require more processing capabilities and more time to parser it. By contrast, the "lightest" body is the CSV response, however, this format is not enough flexible for describing data, compared to JSON and XML. Finally, due to the nature of data, Xively offers to reply the requested data directly in a graphic PNG format.

```
{
    "id":"example",
    "current_value":"500",
    "at":"2013-05-06T00:30:45.694188Z",
    "max_value":"500.0",
    "min_value":"333.0",
    "version":"1.0.0"
}
```

Figure 60: Response in JSON format for reading a single Datastream (Xively_e, 2013)

```
<?xml version="1.0" encoding="UTF-8"?>
<eeml
    xmlns="http://www.eeml.org/xsd/0.5.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="0.5.1" xsi:schemaLocation="http://www.eeml.org/xsd/0.5.1
http://www.eeml.org/xsd/0.5.1/0.5.1.xsd">
    <environment updated="2013-05-05T19:40:08.859383Z" created="2013-
03-29T15:50:43.398788Z" id="121601"
creator="https://xively.com/users/calumbarnes">
        <data id="example">
            <current_value at="2013-05-
06T00:30:45.694188Z">500</current_value>
            <max_value>500.0</max_value>
            <min_value>333.0</min_value>
        </data>
    </environment>
</eeml>
```

Figure 61: Response in XML format for reading a single Datastream (Xively_e, 2013)

```
2013-05-06T00:30:45.694188Z,500
```

Figure 62: Response in CSV format for reading a single Datastream (Xively_e, 2013)

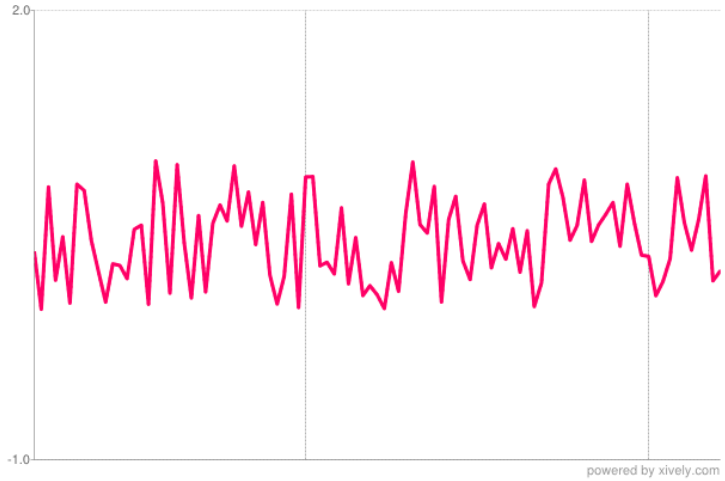


Figure 63: Response in PNG format for reading a single Datastream (Xively_e, 2013)

3.6. Carriots

3.6.1. Introduction

Carriots (Carriots, 2013) stands for Carry the Internet Of ThingS, and aims to provide a M2M platform for the easy, fast and cheap integration of individual devices of amateurs and / or complex systems of companies. Carriots platform was coined in 2010 to manage data from hundreds of millions of devices, and in October 2013 it will be launched the commercial version. The beta version gives developers accounts for free for managing up to 10 devices.

Figure 64: Carriots logo (Carriots, 2013)

3.6.2. Carriots hierarchy model

Following IoT philosophy, Carriots platform allows individual devices and systems communicating among them in any combination (e.g. device to device, device to system, system to device and system to system). No matter the complexity of the system or the simplicity of the device: Carriots platform see all as "devices".

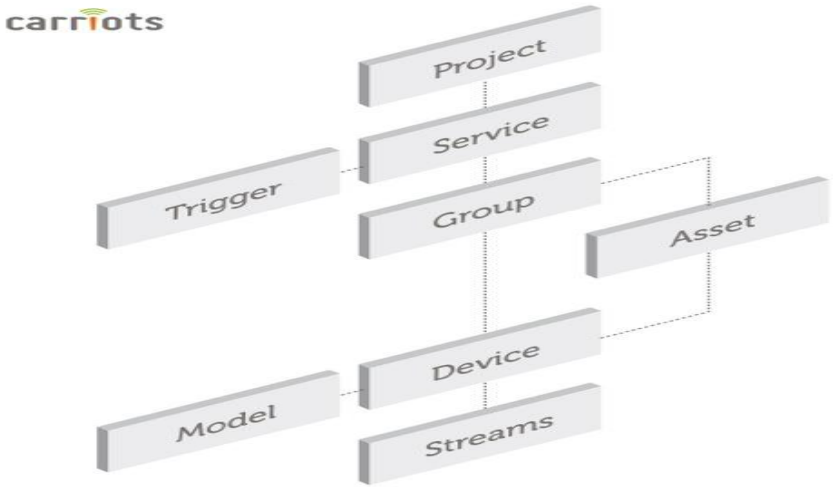


Figure 65: Carriots hierarchy (Carriots_b, 2013)

The Carriots hierarchy is “Project” -> “Service” -> “Group” -> “Device”. One developer owns a Carriots account. Within the account, the developer can create several generic projects (e.g. an account could be for a city and it could have projects like “environmental monitoring” or “traffic management”). Each project will provide services (e.g. following the mentioned example, the project “environmental monitoring” could provide services related to “air quality” and “ground conditions”). These services will be fed by data called “Streams” and provided to the platform through “Devices”. Devices can group features that can be grouped in their “Model”.

Carriots platform offers a wide range of choices such as transferring services’ data received to external systems via triggers associated to such services. In addition, Carriots allows monitoring data streams with “listeners” that wait for an event to occur which might be associated to any level of the hierarchy.

3.6.3. Carriots API

The heart of the Carriots technology platform is its Carriots SDK; internal libraries based on Groovy scripting which users import for developing software in external devices to execute listeners and rules. Devices interact with the platform through the Carriots REST API. The HTTP requests and responses support JSON and XML formats.

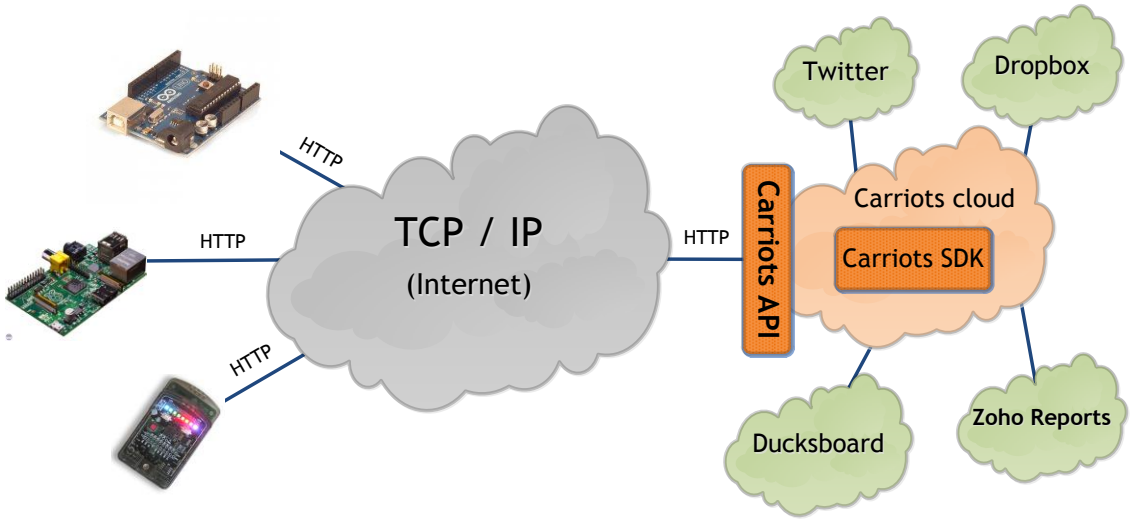


Figure 66: Overview of Carriots platform

Carriots can control user’s Twitter account or Dropbox account even it can send data from the platform towards Ducksboard or Zoho Reports.

The following charts specify in detail most of the HTTP requests parameters for managing the resources of the platform: projects, services, groups, assets, devices, models, data streams, status data streams, triggers, rules, listeners and alarms.

Data streams, triggers, rules, listeners and alarms.

HTTP request	Description
200 OK	Request of the action performed was successfully processed
400 Bad Request	Request is not valid likely due to a malformed JSON body.
401 Unauthorized	Error in the actions "Create a data stream" and "Create a status stream"
404 Not found	The API does not match the requested resource with any of the available ones.
503 No server error	Carriots' services are unavailable

Chart 26: HTTP status codes of Carriots API

Projects

Action performed	Method	Endpoint
Create a project	POST	/projects
Show a project	GET	/projects/[id_project]
List projects, max 1000	GET	/projects
Update project value	PUT	/projects/[id_project]
Delete a project	DELETE	/projects/
Show services	GET	/projects/[id_project]/services/[id_service]/
List services, max. 1000	GET	/projects/[id_project]/services/

Chart 27: Actions performed to Projects in Carriots API

Services

Action performed	Method	Endpoint
Create a service	POST	/services/
Show service data	GET	/services/[id_service]
List services, max 1000	GET	/services/
Update a service	PUT	/services/[id_service]
Delete a service	DELETE	/services/id_service
Show a group from a service	GET	/services/[id_service]/groups/[id_group]/
List group	GET	/services/[id_service]/groups/

Chart 28: Actions performed to Services in Carriots API

Groups

Action performed	Method	Endpoint
Create a group	POST	/groups/
Show a group	GET	/groups/-- ID_developer --/
List a group, max 1000	GET	/groups/
Update a group	PUT	/groups/-- ID_developer --/
Delete a group	DELETE	/groups/-- ID_developer --/
Show an asset from a group	GET	/groups/-- Group ID_developer --/assets/-- Asset ID_developer --/
List assets from a group, max 1000	GET	/groups/-- ID_developer --/assets/
Show device from a group	GET	/groups/-- Group ID_developer --/devices/-- Device ID_developer --/

Chart 29: Actions performed to Groups in Carriots API

Assets

Action performed	Method	Endpoint
Create an asset	POST	/assets/
Show an asset	GET	/assets/-- ID_developer --/
List assets, max 1000	GET	/assets/
Update an asset	PUT	/assets/-- ID_developer --/
Delete an asset	DELETE	/assets/-- ID_developer --/
Show a device from an asset	GET	/assets/-- Asset ID_developer --/devices/-- Device ID_developer --/
List devices from an asset	GET	/assets/-- ID_developer --/devices/

Chart 30: Actions performed to Assets in Carriots API

Devices

Action performed	Method	Endpoint
Create a device	POST	/devices/
Show device information	GET	/devices/-- ID_developer --/
List devices, max 1000	GET	/devices/
Update a device	PUT	/devices/-- ID_developer --/
List of device's configuration files, max 1000	GET	/devices/-- ID_developer --/deviceconfigs/
Download device config file content	GET	/devices/-- Device ID_developer -- /deviceconfigs/-- File name --/-- File version --/
List streams from a device	GET	/devices/-- Device ID_developer --/streams/
Delete streams	DELETE	/devices/-- Device ID_developer/streams/

Chart 31: Actions performed to Devices in Carriots API

Models

Action performed	Method	Endpoint
Create a model	POST	/models/
Show models	GET	/models/-- ID_developer --/
List models, max 1000	GET	/models/
Update a model	PUT	/models/-- ID_developer --/
Delete a model	DELETE	/models/-- ID_developer --/

Chart 32: Actions performed to Models in Carriots API

Data streams

Action performed	Method	Endpoint
Create a data stream	POST	/streams/
Show a data stream	GET	/streams/-- ID_developer --/
List data streams, max 1000	GET	/streams/
Delete a data stream	DELETE	/streams/-- ID_developer --/
List status stream, max 1000	GET	/streams/?_t=sta
Delete status stream	DELETE	/streams/-- ID_developer --/

Chart 33: Actions performed to Data streams in Carriots API

Status of data streams

Action performed	Method	Endpoint
Create a status stream	POST	/status
Show status stream	GET	/streams/-- ID_developer --/
List status stream, max 1000	GET	

Chart 34: Actions performed to Status data streams in Carriots API

Triggers

Action performed	Method	Endpoint
Create a trigger	POST	/triggers/
Show a trigger	GET	/triggers/-- ID_developer --/
List triggers, max 1000	GET	/triggers/
Update a trigger	PUT	/triggers/-- ID_developer --/
Delete a trigger	DELETE	/triggers/-- ID_developer --/

Chart 35: Actions performed to Triggers in Carriots API

Rules

Action performed	Method	Endpoint
Create a rule	POST	/rules/
Show rules	GET	/rules//
List rules, max 1000	GET	/rules/
Update a rule	PUT	/rules//
Delete a rule	DELETE	/rules//

Chart 36: Actions performed to Rules in Carriots API

Listeners

Action performed	Method	Endpoint
Create a listener	POST	/listeners/
Show a listener	GET	/listeners//
List listeners, max 1000	GET	/listeners/
Update a listener	PUT	/listeners//
Delete a listener	DELETE	/listeners//

Chart 37: Actions performed to Listeners in Carriots API

Alarms

Action performed	Method	Endpoint
Create an alarm	POST	/alarms/
Show an alarm	GET	/alarms/-- ID_developer --/
List alarms, max 1000	GET	/alarms/
Update an alarm	PUT	/alarms/-- ID_developer --/
Delete an alarm	DELETE	/alarms/-- ID_developer --/

Chart 38: Actions performed to Alarms in Carriots API

The two figures below show the HTTP body requests and their respective HTTP responses in JSON format of the actions "creating a project" and "creating a device".

```
{
  "name":"defaultProject",
  "enabled":true,
  "id_customer":"@aesaes",
  "description":"default project",
  "id_developer":"defaultProject@aesaes"
}
```

Figure 67: Request in JSON format of "creating a project" (Carriots_c, 2013)

```
{
  "code":"2001",
  "message":"Project created",
  "details":{
    "name":"defaultProject",
    "enabled":true,
    "id_customer":"50d25c0a5c5d75ac1a000006",
    "description":"default project",
    "id_developer":"defaultProject@system",
    "created_at":1355971293,
    "owner":"system",
    "_id":"50d27add5c5d75b81a000013"
   }
```

Figure 68: Response in JSON format of "creating a project" (Carriots_c, 2013)

```
{
  "type":"Arduino",
  "id_group":"Grupo@carriots",
  "networking":
  {
      "telephone_number":"620145282",
      "sim_card":"234582385934578964",
      "carrier":"vodafone",
      "type":"gprs",
      "conn_address":null
  },
  "name": "newdevice",
  "frequency_stream": 5,
  "firmware":"db01.v2.7.4.0.0.4_BEEEE",
  "sensor":"thermometer",
  "description":"Tarjeta Arduino",
  "frequency_status":5,
  "checksum":"8e0f1692c01e3333d0cdefe6840c5d61ffb7e3b7",
  "enabled":true,
  "time_zone":"Europe/Madrid"
}
```

Figure 69: Request in JSON format of "creating a device" (Carriots_d, 2013)

```
{
"code":"2001",
"message":"Device created",
"details":
   {
     "type":"Arduino",
     "id_group":"5075366928ddcbe564000055",
     "networking":{
                     "telephone_number":"620145282",
                     "sim_card":"234582385934578964",
                     "carrier":"vodafone",
                     "type":"gprs",
                     "conn_address":null
                    },
      "name":"newdevice",
      "frequency_stream":5,
      "firmware":"db01.v2.7.4.0.0.4_BEEEE",
      "sensor":"thermometer",
      "description":"Tarjeta Arduino",
      "frequency_status":5,
      "checksum":"8e0f1692c01e3333d0cdefe6840c5d61ffb7e3b7",
      "enabled":true,
      "time_zone":"Europe/Madrid",
      "id_developer":"newdevice@carriots",
      "status":"ok",
      "created_at":1355967817,
      "owner":"carriots",
      "_id":"50d26d495c5d75b81a00000d"
   }
}
```

Figure 70: Response in JSON format of "creating a device" (Carriots_d, 2013)

PART II: CASE STUDY

CHAPTER 4: **nano Service-Oriented Middleware (nSOM)**

CHAPTER 5: **SYSTEM ANALYSIS**

CHAPTER 4: nano Service-Oriented Middleware (nSOM)

4.1. Introduction

The second part of this Final Project Dissertation focuses on nSOM, a middleware developed by the research group GRyS (Grupo de Redes y Servicios de próxima generación) of the Universidad Politécnica de Madrid (UPM) in Madrid (Spain). It aims to allow communication among heterogeneous and constrained nodes which belong to a network. nSOM middleware abstracts the complexity of hardware platform to developers of applications.

Sun SPOT mote is the hardware platform selected which embeds the nSOM middleware, and provides services programmed in applications. The platforms belong to a WSAN and their services are available for any user with an Internet connected device.

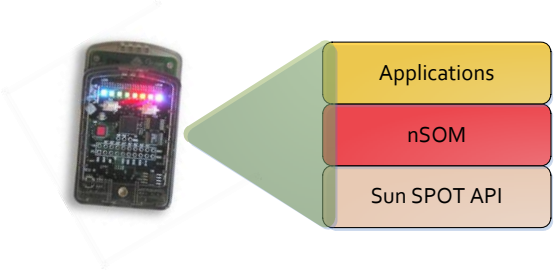


Figure 71: nSOM embedded in Sun SPOT mote

nSOM architecture follows the principles of Service-Oriented Architecture and is responsible of managing the requests to the services from users. However, in the near future nSOM will incorporate functionalities such as events that alerts automatically to users.

This chapter begins with a comparison between Service-Oriented Architecture and Event-Driven Architecture, and follows with a detailed description and specification of nSOM internals such as software components, the interaction of the components, the Device Manager component specification, and finishes with the extended nSOM protocol.

4.2. Service Oriented Architecture versus Event-Driven Architecture

Service-Oriented Architecture (SOA) came up in 2003 next to the globalization of the Internet and was introduced by Roy Schulte as a paradigm that gives design principles for software architectures (Erl, 2008) (Davis, 2010). Services are the heart of the paradigm and all components are built around them. They can be understood as a logic solution that customers request via indirectly communication to the service provider. SOA defines the loosely coupled interaction among services and their components. It establishes the guidelines for the relationship among services, customers and service providers.

Regarding other solutions for distributed applications, SOA introduces the novelty of loose coupling of underlying systems which are connected together, and provides abstraction for both customers and service providers. Usually there is an intermediary between customers and service providers which manages the exchange of messages for requests and responses. The intermediary is a middleware that makes the tasks related to publishing services and is often associated to a database (registry) where stores a list of services linked to their corresponding services providers. Thanks to the middleware, neither customers require knowing where the service providers are nor services providers "who" requests their services.

- **Service providers** are those which offer services and publish them "where" customers can find them.
- **Service consumers** are the software components that interact with services via request messages to "where" services are published.

The figure below illustrates the main actors in SOA.

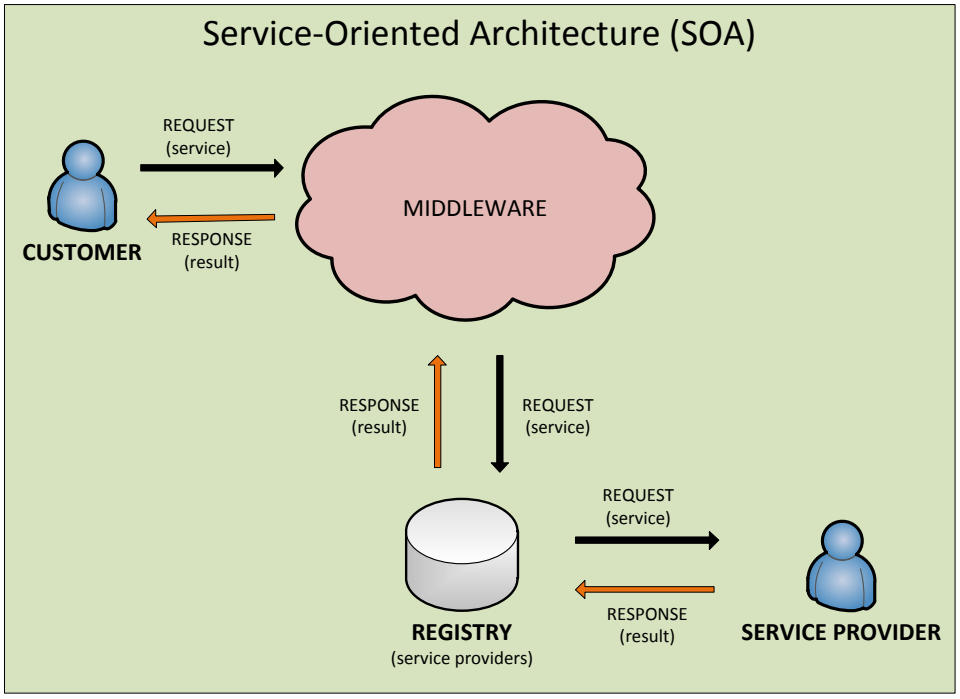


Figure 72: Service-Oriented Architecture paradigm

However, although SOA makes the customers and services providers decoupled, it does not define mechanisms if circumstances change, so customers must poll for changes requests. Event-driven architecture (EDA) encompasses guidelines for detecting changes in the environment (events) and can react to them via sending alerts to event customers previously subscribed to those events (Taylor, et al., 2009). This change in the data flow gives more flexibility to event customers, so they do not have to be polling continuously to detect a change.

EDA architecture is formed of certain components which have to be organized to transmit messages when detecting events, and react to them in real time when receive such messages notifying events. EDA must include mechanisms to detect events, a protocol to handle messages when detecting events, processing and reacting messages. These functionalities are associated to the several architectural names such as event producers, event managers, or event consumers, among others. These components are applications programmed to publish, subscribe, or take other actions when events triggered.

- **Events** are "notably" actions that can occur any time and imply a change in state. But not all actions are events, so they have to be defined according to specific rules.
- **Event producers** could adopt many different forms due to the fact that an event can be any kind of defined action. Most are pieces of dedicated software programs which monitor a piece of hardware (sensors) continuously, and according to a set of defined rules, decide if an event has been produced or not.
- **Event manager** is a piece of software which decides the next action to take when an event has occurred (reactions). An example of reaction could consist in an automated notification sent either to applications or people.

The table below establishes an analogy between SOA and EDA main actors.

SOA or SOA 1.0	EDA or SOA 2.0
Service provider	Event producer
Customer	Event consumer
Middleware service-oriented	Middleware event-driven

Chart 39: Analogy among SOA and EDA main components

4.3. nSOM software components

This section introduces the internal software architecture of nSOM. It comprises a combination of modular software elements that interact among them in an organized manner. Their core software elements are the nSOMContainer subsystem, Service Manager component, and the nSOMProtocol component. In addition, in order to enrich the development of applications, nSOM implements extra software components used by applications to offer services. nSOM provides several components to achieve this goal such as "Security" or "Device Manager", among others.

Next figure presents the relationship of the main software components of nSOM.

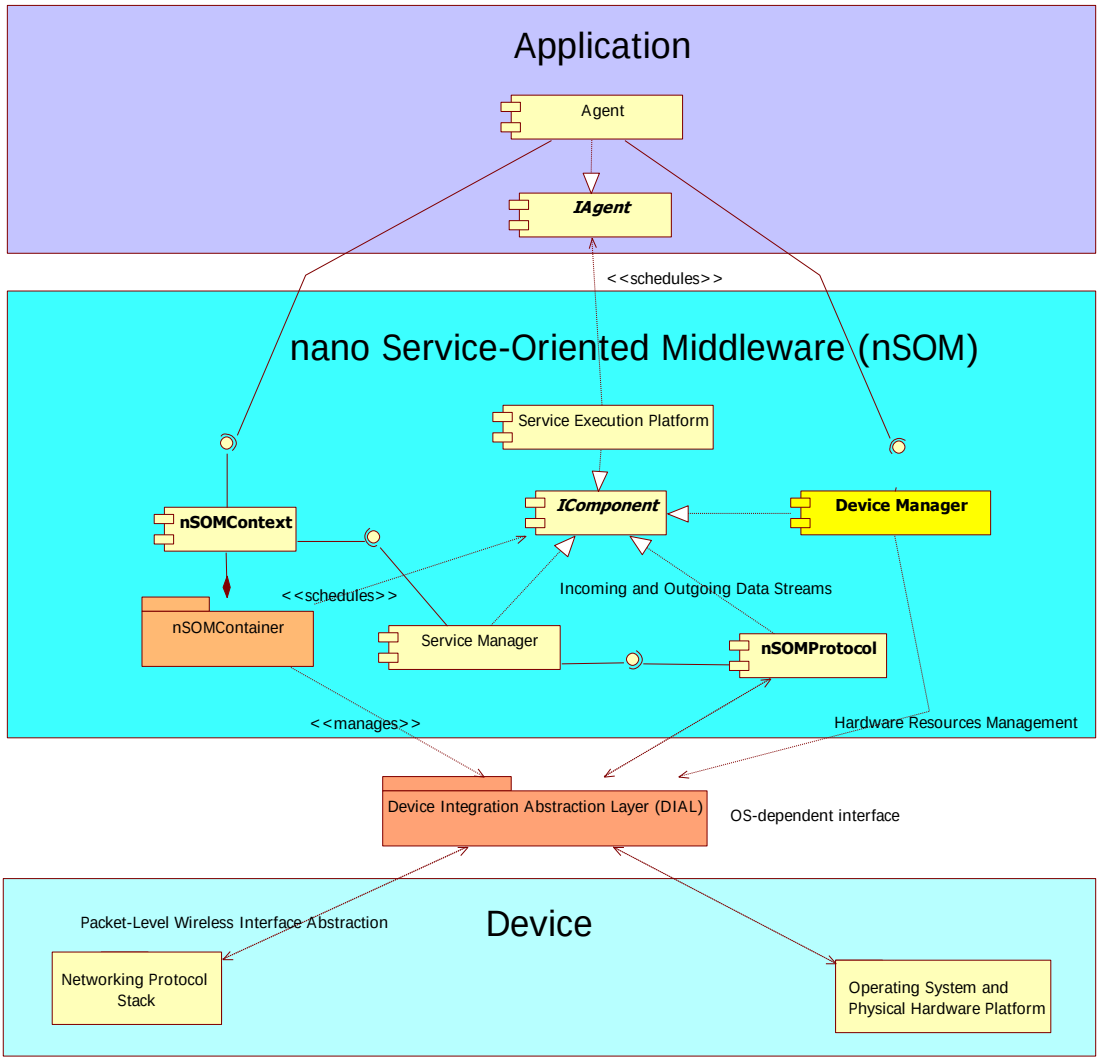


Figure 73: nSOM software components relationship

- **nSOMContainer** could be seen as the background of the middleware which "contains" the components. It performs the organized deployment of every component in the architecture, manages the decoupled interaction among components and their life cycle (scheduling); it initialize them with their "context", and make them to start running, stopping or pausing. It uses the nSOMContext to store applications' services in an internal repository of the node.
- If the nSOMContainer was the scheduler of the middleware's components, the **Service Execution Platform** plays the analogous role in the application layer. Applications (agents) are treated as components which belong to the application layer; the Service Execution Platform makes them to start, stop or pause. In addition, Service Execution platform allows deploying more than one application on the node.

- **Service Manager** orchestrates the composition of the HELLO message payload in JSON format that is sent when the node powers on. The next section describes the interaction model between the main components when the device turns on.
- **nSOM Protocol** abstracts the developer of the underlying networking protocol stack of the device. It encapsulates tasks regarding sending and receiving nSOM messages to and from other nodes (publishing applications' services and responding for services requests).
- The **DIAL** refers to a software layer that controls the underlying hardware platform. It hides to nSOM hardware dependent details such as networking protocols stack (Ethernet, WiFi, 3G, etc.), operating system, and circuitry such as microprocessor communications mechanisms, I/O interfaces, etc. It is dependent on the device, so it cannot be ported without changing its code interface. nSOM uses DIAL for sending and receiving packets through radio interface and getting raw data from sensors.

4.4. nSOM components interaction

The behavior of nSOM when a generic node powers on is presented in the sequence diagram. This interaction model describes how the main internal components of the nSOM communicate with each other. When the node powers on, it must show its availability via sending the HELLO message to another node in order to "communicate" the services offered.

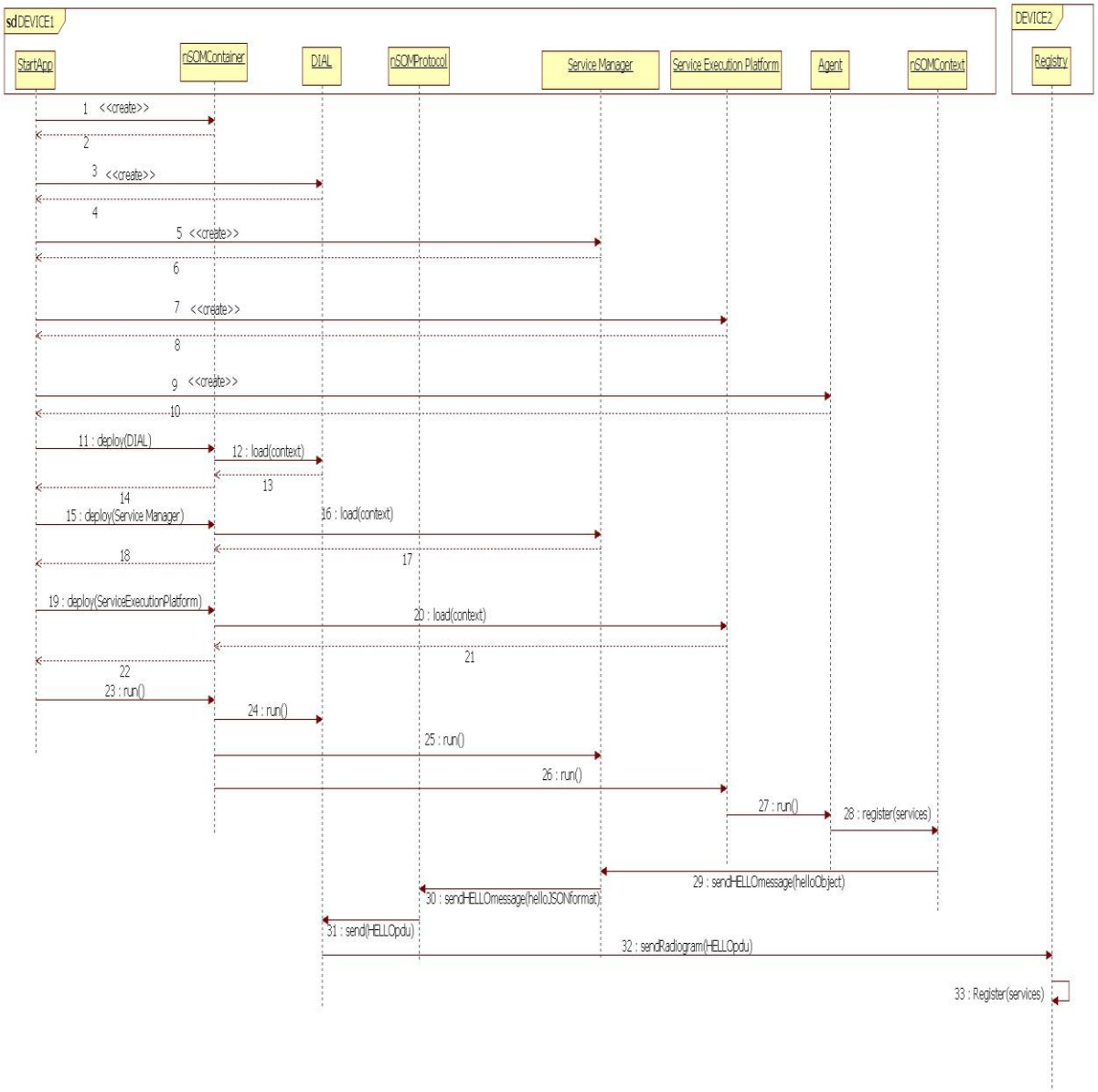


Figure 74: nSOM components sequence diagram

Interactions among the components can be divided in three primary phases. The first corresponds to creating instances of each one. The second is about the deployment of each component in the nSOMContainer. In the third, nSOMContainer performs the executing of all components.

In the first phase (1 - 10) all components are instantiated to allow deploying them in the nSOMContainer. Agents are also instantiated to deploy them in the Service Execution Platform. Due to the role of scheduler of the nSOMContainer, it is the first element instantiated. The nSOMContainer instantiation implies of creating the nSOMContext which will be the interface for the agents. These agents will store its services description in an internal repository of the node allowing the rest of components to access them.

The second phase (11 - 22) comprises the deployment of the components in the nSOMContainer. nSOMContainer "initializes" each component with the context to allow them to access the repository where the services offered by the agents.

The third and the last phase (23 - 31) is the start up execution of the components. It can be appreciated in the sequence diagram how the nSOMContainer manages such executions with the run method. Note the last component to be launched is the Service Execution Platform which performs the execution of each agent deployed on it, similarly as the nSOMContainer does with every components of the nSOM. The deployment of agents in the Service Execution Platform produces a chain interaction among components in order to make up the nSOM PDU of the HELLO message; running agents serves for passing their services description to the nSOMContext which stores them in the repository. Then the Service Manager will access to them and adapt the services description data in JSON format. Finally, the nSOMProtocol creates the HELLO PDU and passes it to the DIAL for sending it over the radio interface (in our case is HELLO PDU is encapsulated with the Radiogram protocol but it could be any other such as Ethernet, WiFi, 3G, etc.).

HELLO message is sent in broadcast mode and is processed by devices that play the role of "Registries" (DEVICE2). Services stored in the Registry will be associated to the proper device (DEVICE1) and the agent which provide such services. Then, any external customer (e.g. a being human) can access to them via polling method.

Once the node has powered on and their services are registered, it gets in "listening" mode indefinitely. Then it is ready for receiving requests for their services.

4.5. Device Manager component specification

Device Manager component aims to collect data from the device which belongs to the Internet of Things in order to monitor it and enable proactive fault detection. This component will help to provide all kind of information to an external system (e.g. a Sun SPOT mote uploading their status information to a cloud platform) in a way that it is easy to build reports and dashboards which show the most common device information, including device population, status, hardware resources, location and other useful data. This component belongs to the nSOM middleware and has been designed to be used for applications (agents) deployed in nodes. This is a generic component used within this Final Project Dissertation with Sun SPOT motes (described them in chapter 2).

Firstly, an overview of the software internals and its relationship is exposed in the next class diagram expressed in UML (Unified Modeling Language). For clarity and space paper constraints, attributes and methods have been omitted.

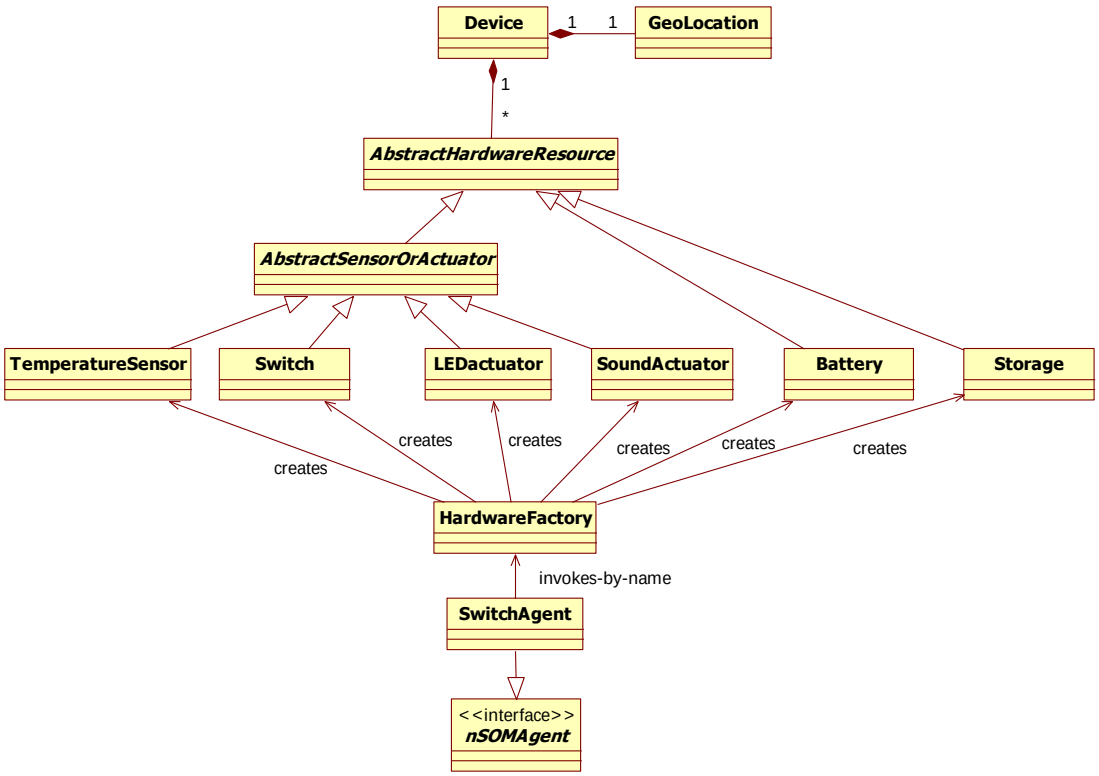


Figure 75: Device Manager diagram class

The design principles are based in the fact that every device connected to the Internet of Things is composed of "hardware" (e.g. sensors, actuators, etc.) and is physically located in a specific place (e.g. inside a house, outside in a park, flying on an aircraft, etc.). It is focus on three core parts; the overall features of the device itself (Device class), its geographic location (GeoLocation class), and its hardware resources (AbstractHardwareResource class).

A device can be seen as a single unit or as a composition of multiple devices. For instance, the hardware platforms analyzed in chapter 2 contains various types of integrated sensors (e.g. humidity, temperature), actuators (e.g. LED) and several expansion pins to amount external circuitry. Then we can see the node as a whole device or like a composition of multiple sensors, actuators, etc., acting as individual devices. To accomplish the design of this software component, I assume that a "device" is the technical artifact which provides an interface between digital and physical world and they are composed of various "hardware resources" mentioned before.

Devices must have communication capabilities in order to connect to the Internet (e.g. WiFi, 802.15.4, 3G, etc.), minimum computation capabilities in order to process data exchanged with external systems, and storage to support firmware or software and taking smart actions. Other important capabilities are power resources to give operational autonomy to the device.

4.5.1. Device class and GeoLocation class

The Device class groups general features of a generic device such as an unique identifier for distinguishing from others, a list of hardware resources that it contains (e.g. sensors, actuators, etc.), the type of the device (e.g. smart phone, wireless sensor device, desktop computer, etc.), the role that the device plays (e.g. a mote for harvesting the temperature and another for measuring the humidity), manufacturer (e.g. Sun SPOT motes, Waspmote motes, etc.), mobility (e.g. smartphone or fixed mote) and its geographic location. Every device can be characterized with these data. However, there are millions of devices, and it could happen that these attributes did not represent special features at all; in this case the attribute "extraInfo" should be used.

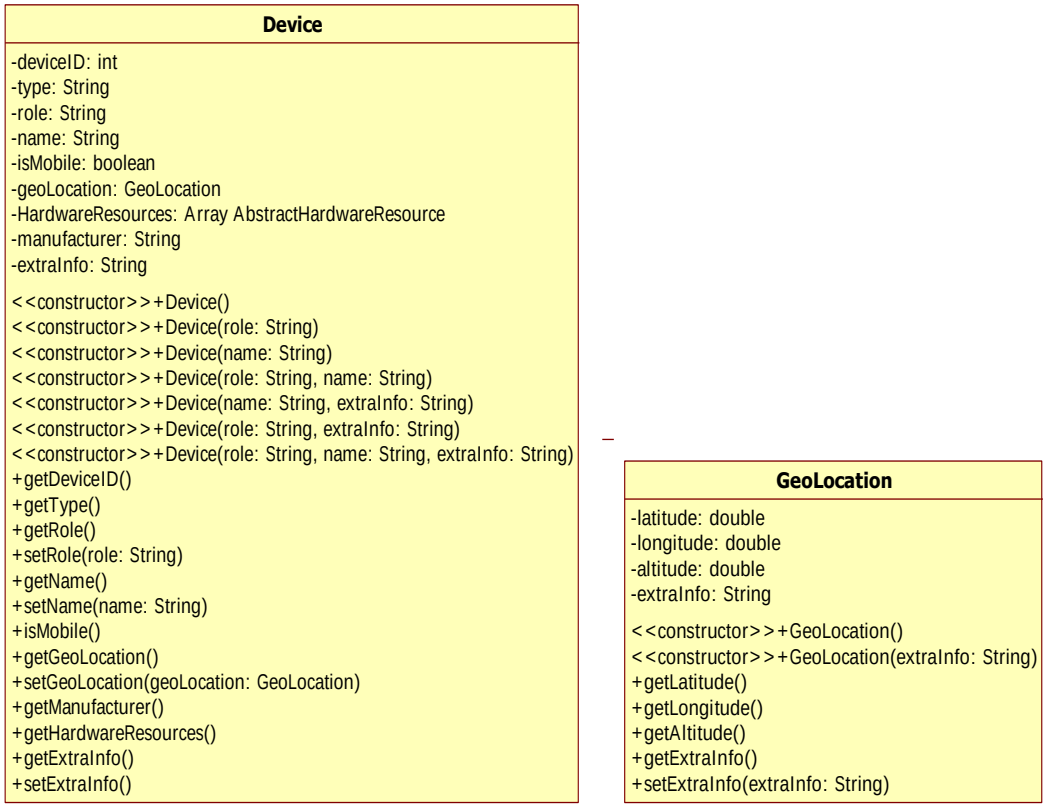


Figure 76: Device class and GeoLocation class

Note that GeoLocation class also contains the private attribute "extraInfo" accessible through public method "getExtraInfo" and modifiable with "setExtraInfo". This attribute is useful when the device does not support GPS and is not possible to extract neither latitude nor longitude. So with this attribute can express the place and more detailed comments.

4.5.2. AbstractHardwareResource class

The AbstractHardwareResource class is an abstract dependent on device class. As stated in chapter two, devices differ in hardware resources such as sensors, connectors, serial communication interfaces, etc. This class is intended to be a generic one that can be adapted to a specific device according to concrete requisites. For example, within electronic design level it

could be interesting the internal circuitry of the device, so the hardware resources could be chipsets, resistors, capacitors, and so on. In our case study, we are interested in the information that devices can provide, so the highlighted hardware resources are sensors and actuators. In the near future, we have planned to include the battery in order to check its level and act when it is running out, and storage to get information about storing capabilities of devices.

This abstract class contains the common attributes and methods that every concrete resource inherits. The class diagram shows the specific hardware resources used within our case study; temperature sensor (TemperatureSensor class), Swith (Swith class), LED (LEDactuator class) and sound (SoundActuator class).

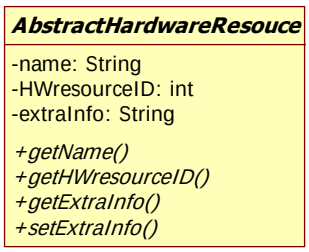


Figure 77: AbstractHardwareResource class

4.5.3. AbstractHardwareResource's child classes

A part of the inherited attributes and methods of super class, every concrete hardware resource will be characterized with their concrete attributes and methods specially designed to extract data of that resource. For example, temperature sensor will focus on harvesting temperature data. Based on this basic data, more sophisticated methods can be added, such as the date of the first measure or the last measured value. This example can be applied to the rest of hardware resources.

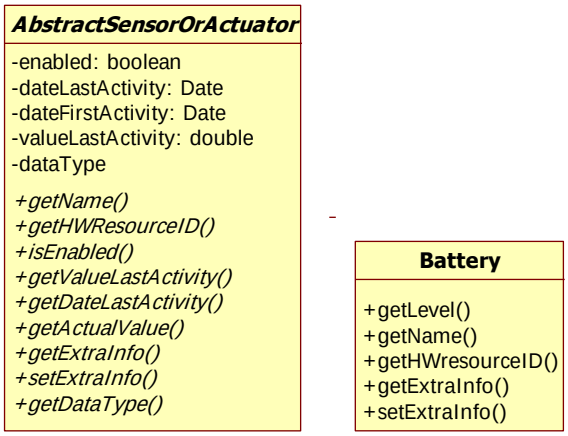


Figure 78: AbstractSensorOrActuator class and Battery class

During the design process, I have concluded that there is an analogy between sensors and actuators. The result is the creation of the abstract class "AbstractSensorOrActuator" which groups their common attributes and methods. Although the data flow of each group is the opposite, both can be featured with the attributes regarding their "activities" and "data type". For sensors, activities refer to collect data, that is to say, input data (e.g. temperature sensor). For actuators, they mean output data (e.g. LED actuator). On the other hand, the data type specifies the kind of input or output data (e.g. temperature sensor measures degrees, and LED actuator turn on or off with digital or Boolean programmatic feature).

4.5.4. HardwareFactory class

The HardwareFactory class is very useful for applications to invoke and create instances of concrete hardware resources in an organized and comfortable manner. The class diagram shows how the application SwithAgent invokes this class to create an instance of Switch class and control the underlying hardware using the methods.

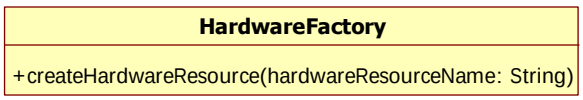


Figure 79: HardwareFactory class

4.5.5. SwitchAgent class and nSOMAgent interface

SwithAgent class represents the application that uses the Device Manager component to accomplish its purpose. In this particular case, it invokes the HardwareFactory class to create the objects of the hardware resources that it needs such as Switch, LEDs and sound.

This application monitors the switch number one, then when the last one is pressed, two actions occurred; the LED panel turns on and sounds a "beep". This application informs to the requesters if the switch number one has been pressed or not. It is has been pressed it will response "true", otherwise "false" is returned.

On the other hand, SwithAgent is a concrete agent that inherits part of its methods that are common to all agents from the interface nSOMAgent. The inherited methods are exposed in the interface nSOMAgent shown below.

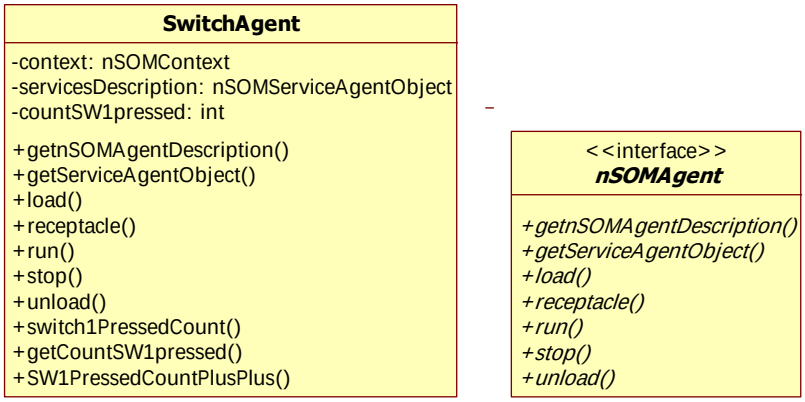


Figure 80: SwitchAgent and nSOMAgent class

4.6. nSOM protocol

nSOM protocol has defined a generic PDU to exchange data between nodes. The PDU is structured with a header and a payload. The header specifies the type of protocol of the lower layer in which it is encapsulated, the type of message transmitted, and its length. The payload consists of fields which are key-value pairs.

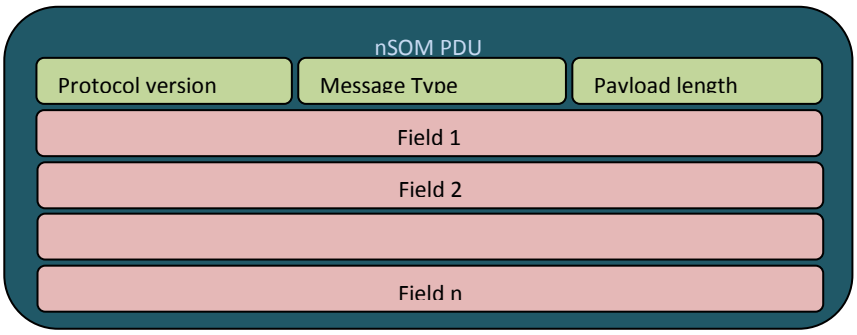


Figure 81: Generic nSOM PDU

Currently, nSOM protocol has 12 messages defined (Rodríguez-Molina, 2012) which were conceived to satisfy the SOA requirements. This section aims to expand the protocol with the detailed specification of 7 new messages to make nSOM architecture tend to more dynamic SOA 2.0 paradigm.

The table below shows a summary of the messages specified.

HELLO Request	**HELLO Response**
PUBLISH EVENT Request	PUBLISH EVENT Response
SUBSCRIBE to EVENT Request	SUBSCRIBE to EVENT Response
NOTIFY EVENT	

Chart 40: Table of nSOM event-oriented messages

4.6.1. HELLO Request message specification

This is the first message sent by a device when turns on. It aims to announce the presence the device on the network. It also provides information of the device itself and services – demand services and event-oriented services– provided by Service Providers and Event Producers which can be embedded in the device. One message is sent per device.

Origin	**Device**
Target	Event Managers, Registries
Transmission Mode	Broadcast
Payload	Demand services' description, event-oriented services or events' description, device's information.

Chart 41: HELLO Request message

4.6.2. HELLO Response message specification

This message is the response of the HELLO Request message. Due to the heterogeneous environment and the constrained processing capabilities of these devices, it is possible of lost messages with ease. HELLO message is a very important message because is the introduction of the device to the network. If this message is lost, the device does not exist. For these reasons we have estimated convenient to re-define HELLO message and incorporate an "acknowledge". One message is sent per device.

Origin	**Device**
Target	Registry or Event Manager
Transmission Mode	Unicast
Payload	The name of the demand services that have been registered in the Registry, the name of the event-oriented services which have been registered in the Event Manager.

Chart 42: HELLO Response message

This message is "lighter" than HELLO Request; it does neither carry description of services nor device's information.

In the case of demand services, this message guarantees that they are available to any customer for polling in any time, and give the "input" for initializing the services. It should be noted that while the HELLO Response is not processed by the device, the demand services are not running on the device. The reason of this is saving energy: these constrained devices have limited batteries so it is very important to make them work only the necessary. On the other hand, for events, this message also guarantees that they are available to any event consumer to subscribe it. In contrast with demand services, event monitoring does not start running with HELLO

Response message, but with SUBSCRIBE to EVENT message which will be explained in the following sections.

As mentioned above, the combination of constrained processing capabilities and heterogeneous environment can produce lost messages easily, so it might occur that the device does not receive the HELLO Response at first. To solve this problem, the device waits a time for the message. If the time is run out, the device would re-send the message. In the worse case, this process can be repeated several times.

4.6.3. HELLO Request – HELLO Response sequence diagram

The following sequence diagram shows the relationship between the HELLO Request message and HELLO Response when the Device offer demand services.

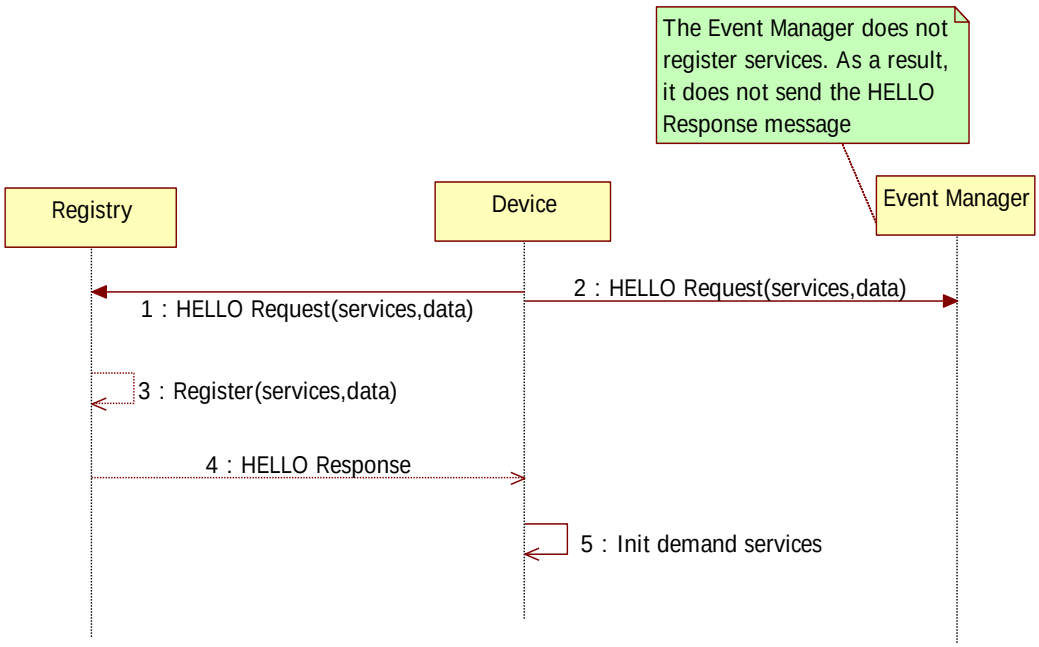


Figure 82: HELLO sequence diagram 1

It should be noted that "data" is referred to the rest of information carried in the message that are not services (device information).

Although in the diagram appears two HELLO Request messages, notice that is only one HELLO Request message sent in broadcast mode and received by all surrounded devices. As the message only carries with demand services, it is only processed by the Register.

This sequence diagram shows the basic elements in order to expose the behavior of the system; however, in real scenarios it could have several Registries, Event Managers and Devices, so the scene would be several devices sending HELLO Request messages and several Registries and Event Managers responding with the HELLO Response message.

The following sequence diagram is analogous to the previous one, and shows the relationship between the HELLO Request message and HELLO Response when the Device offer event-oriented services or events.

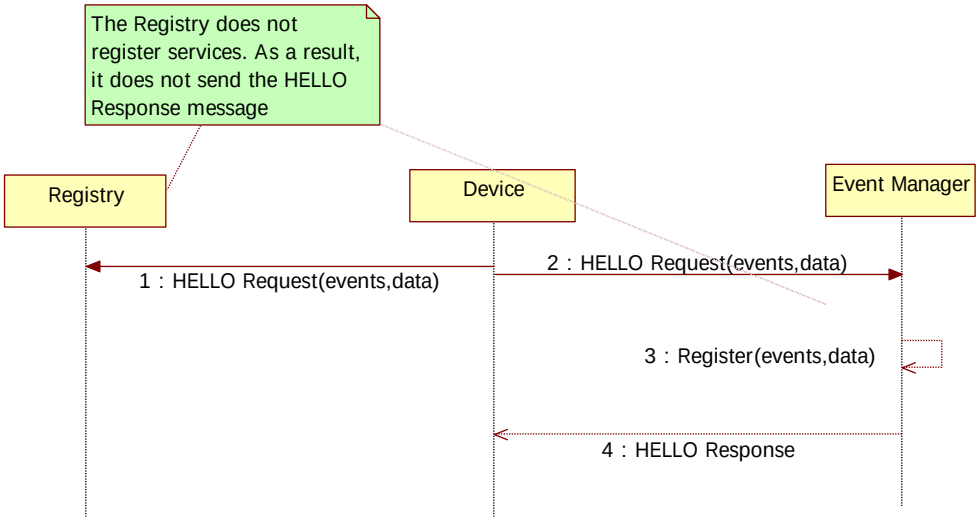


Figure 83: HELLO sequence diagram 2

The following sequence diagram shows the relationship between the HELLO Request message and HELLO Response when the Device offer both events and demand services.

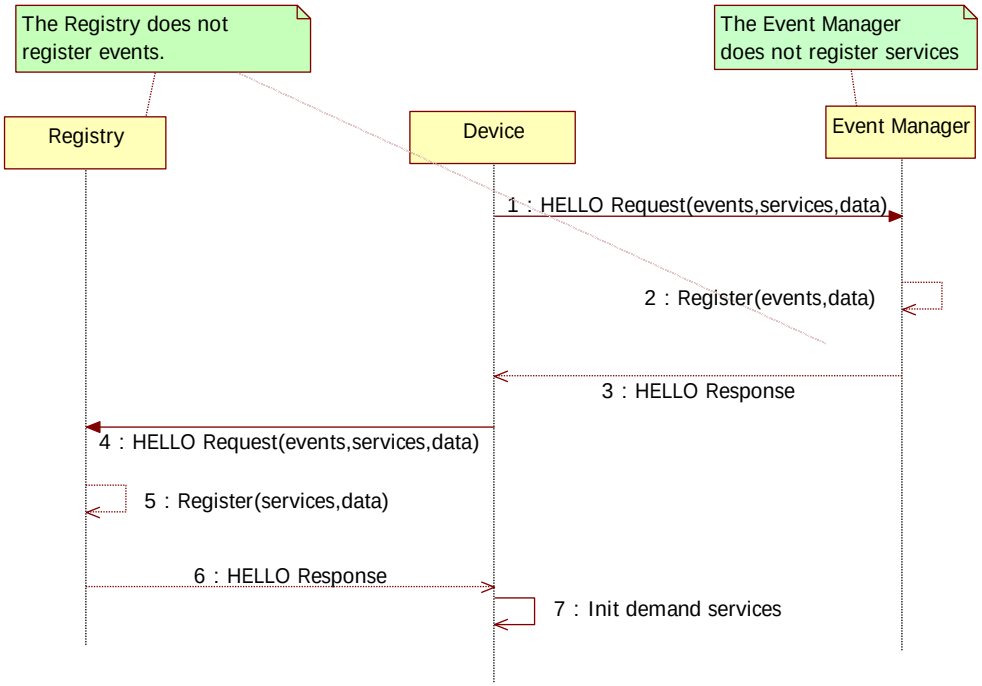


Figure 84: HELLO sequence diagram 3

In order to expose with clarity the sequence diagram, we have grouped the blocks HELLO Request / Response for events first, and the last block for demand services. However, in real scenarios this would be more complex and the sequence of the messages would likely follow a different order; for example, it might occur that the Registry received the HELLO Request before the Device processed the HELLO Response.

4.6.7. PUBLISH EVENT Request message

The Internet of Things' scenarios are framed in a dynamic and evolving environment in which several devices can change roles depending on the circumstances. The goal of this message is to update Event Managers and Registries if the surrounding nodes provide new demand services and / or events.

Origin	**Event Producer, Service Provider**
Target	Event Managers, Registries
Transmission Mode	Broadcast
Payload	Demand services' description, event-oriented services or events' description, device identification.

Chart 43: PUBLISH EVENT Request message

Observers can reach the conclusion of the analogy between the HELLO Request and PUBLISH Request messages, but there are the following differences; in first place, HELLO supports the discovery of the device to the network, while PUBLISH update the role of the device in any time; in second place, the PUBLISH does not carry with device's information as HELLO does, so PUBLISH message is "lighter". However, it carries with device identification so Event Managers and / or Registries have the key to get more details of the device if they needed it. In last place, the PUBLISH message is sent from the new software component installed in the device – which implements the new services and / or events – while HELLO is sent from the Device.

4.6.8. PUBLISH EVENT Response message

This message is the response of the PUBLISH message. The response guarantees that the device's role has been updated in Event Managers and / or Registries, and their new demand and / or services are available to any Customer and / or Event Customer. One message is sent per Event Producer and / or Service Producer.

Origin	**Event Managers or Registries**
Target	Event Producer or Service Provider
Transmission Mode	Unicast
Description	The name of the demand services that have been registered in the Registry and the name of the event-oriented services which have been registered in the Event Manager.

Chart 44: PUBLISH EVENT Response message

It should be noted that this message carries the same payload as HELLO Response does. In the implementation, HELLO Response message is used to cover this message. As a result, the pair of messages PUBLISH Request-PUBLISH Response would be equivalent to PUBLISH Request – HELLO Response. However, in order to avoid misunderstandings, we prefer different names; so when we refer to HELLO Response message, we mean the response of HELLO Request and no of PUBLISH Request.

In the case of demand services, this message guarantees that they are available to any customer for polling in any time, and give the "input" for initializing the services. It should be noted that while the PUBLISH Response is not processed by the device, the demand services are not running on the device. The reason of this is saving energy: these constrained devices have limited batteries so it is very important to make them work only the necessary. On the other hand, for events, this message also guarantees that they are available to any event consumer to subscribe it. In contrast with demand services, event monitoring does not start running with

HELLO Response message, but with SUBSCRIBE to EVENT message which will be explained in the following sections.

The combination of constrained processing capabilities and heterogeneous environment can produce lost messages easily, so it might occur that the device does not receive the PUBLISH Response at first. To solve this problem, the device waits a time for the message. If the time is run out, the device would re-send the message. In the worse case, this process can be repeated several times.

The following sequence diagrams expose the relationship between the PUBLISH Request message and PUBLISH Response. They are analogous to the HELLO Request – HELLO Response sequence diagrams. The sequence diagram below shows the relationship between the PUBLISH Request message and PUBLISH Response when a new Service Provider is deployed in the Device.

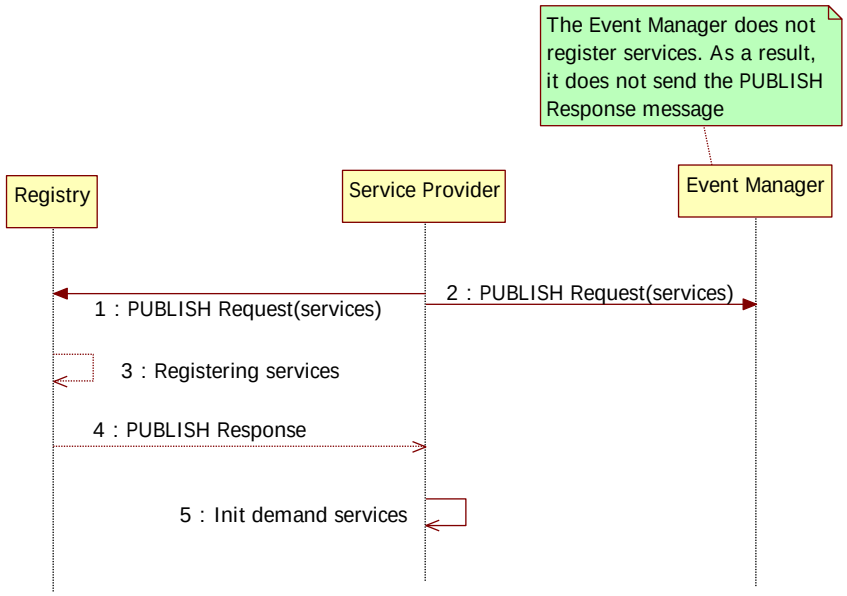


Figure 85: PUBLISH sequence diagram 1

The following sequence diagram shows the relationship between the PUBLISH Request message and PUBLISH Response when a new Event Producer is deployed in the Device.

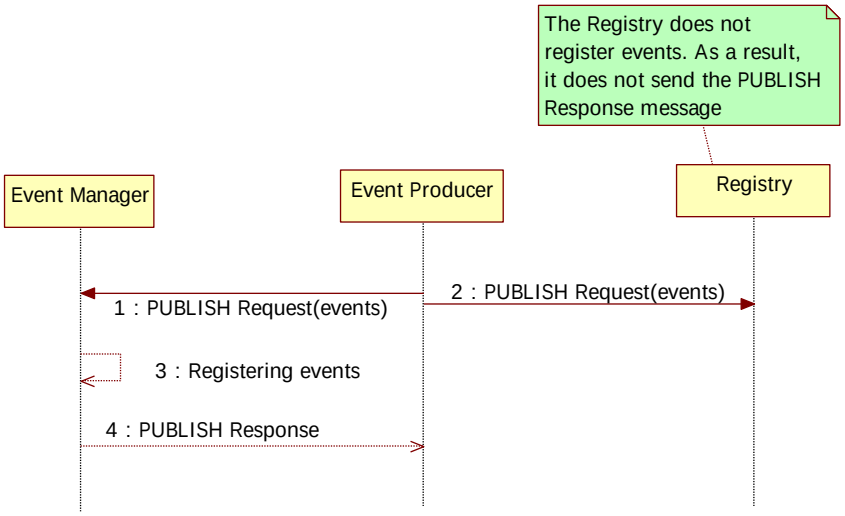


Figure 86: PUBLISH diagram sequence 2

The following sequence diagram shows the relationship between the PUBLISH Request message and PUBLISH Response when a new Service Provider and new Event Producer are deployed in the Device so we assume that the Event Producer and the Service Provider physically located in the same Device.

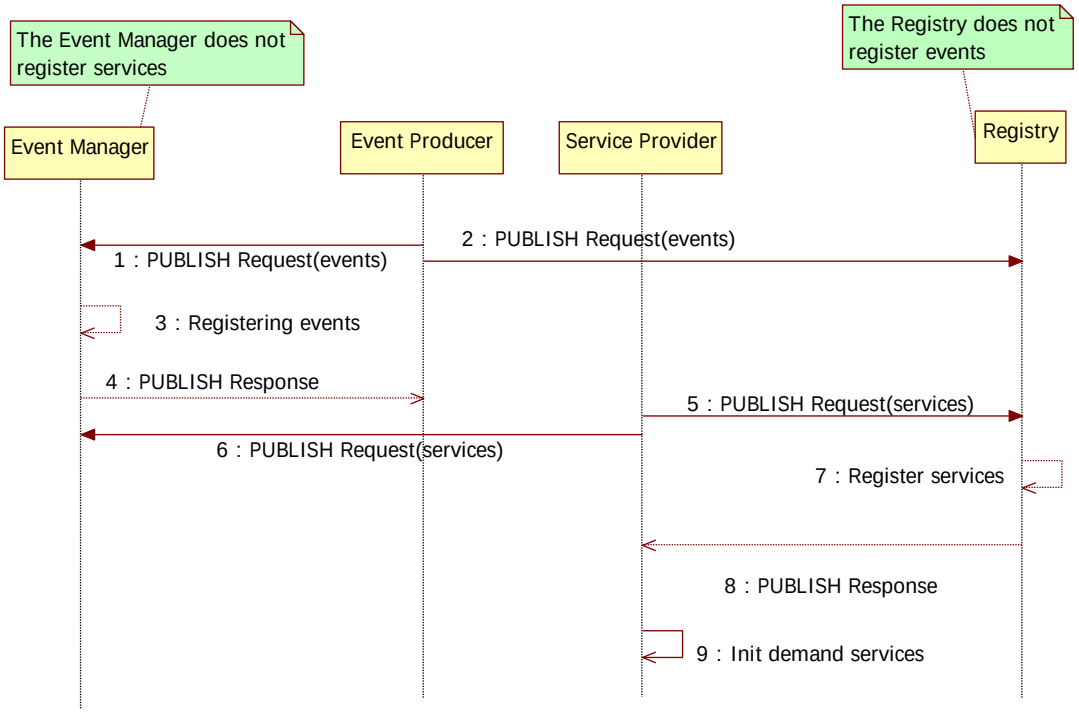


Figure 87: PUBLISH sequence diagram 3

In order to expose with clarity the sequence diagram, we have grouped the blocks PUBLISH Request / Response for events first, and the last block for demand services. However, in real

scenarios this would be more complex and the sequence of the messages would follow the reverse order; it will depend on which was first deployed, the Service Provider or Event Producer. In the previous diagram we assumed that the Event Producer was deployed before Service Provider.

4.6.9. SUBSCRIBE to EVENT Request message

This message carries with the service event-oriented to which the Event Customer would like to be notified if a trigger condition (event) associated to that event-oriented service occurs. The Event Consumer is associated to that event in the Event Manager, and the Event Manager has the relationship between the queried event and the Event Producer.

Origin	**Event Manager**
Target	Event Producer
Transmission Mode	Unicast
Payload	Event oriented services or events' name

Chart 45: SUBSCRIBE to EVENT message

For example, a person (Event Consumer) subscribes to the event-oriented service "notifyPresence"; if anyone enters to the Event Customer's house without permission, this user of the event would receive a NOTIFY EVENT message in his / her Smartphone. If the customer had not been registered, this one never would receive the NOTIFY EVENT message.

4.6.10. SUBSCRIBE to EVENT Response message

This message is the response of SUBSCRIBE to EVENT Request message. It is an intermediary message used to report in real time that the Event Customer will be informed of the event when the condition associated to that event is triggered.

Origin	**Event Producer**
Target	Event Manager
Transmission Mode	Unicast
Payload	Event oriented services or events' name

Chart 46: SUBSCRIBE to EVENT Response message

It should be noted that the payload is the same as the SUBSCRIBE to EVENT Request. The only difference is the directionality of the message; SUBSCRIBE to EVENT Response message goes from the Event Producer to the Event Manager.

The following sequence diagram shows an Event Customer subscribing to an event-oriented service provided by a generic Event Producer. This process is handled through an Event Manager which acts as an intermediary component. As a result, the Event Producer does not require knowing where the Event Customer is.

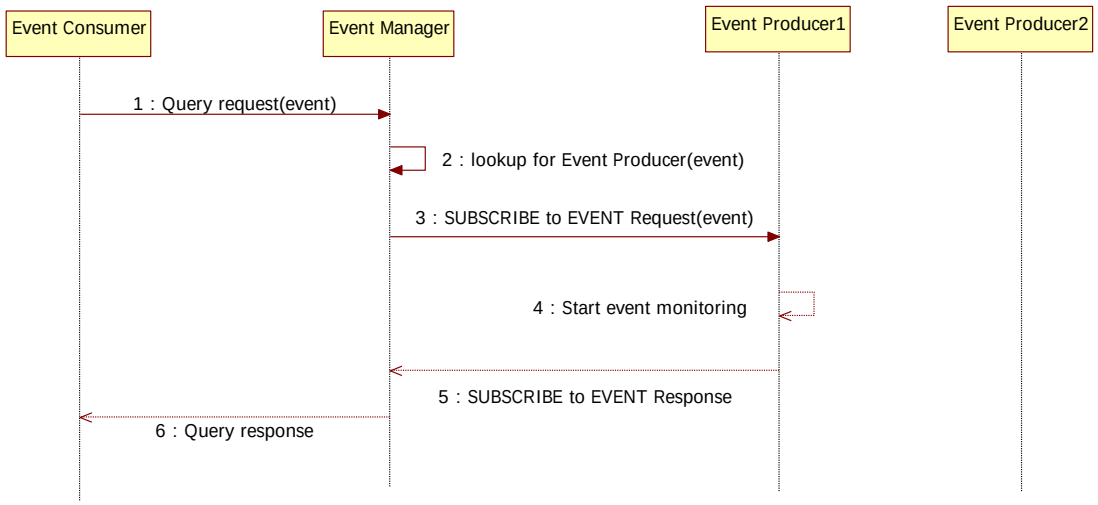


Figure 88: SUBSCRIBE to EVENT sequence diagram 1

In the above diagram we assume that there are two Event Producers who can offer different event-oriented services. The Event Consumer queries to the Event Manager for a specific event (1), and the Event Manager Lookup for an Event Producer which can offer such event (2). It should be highlighted that the Event Producer 1 does not start event monitoring unless it receives a SUBSCRIBE to EVENT Request (3); this situation is analogous to the Service Provider which started running the demand services only when it received the HELLO Response or PUBLISH Response. The reason of doing this is saving energy.

Next figure shows a more general situation in which there are three event-oriented services available for the Event Customer to subscribe and this one select two. First, the Customer selects an event which provides the Event Producer 1 and second, selects another event provided for the Event Producer 2. In a real scenario, it could have more Event Customers which subscribe to the same event.

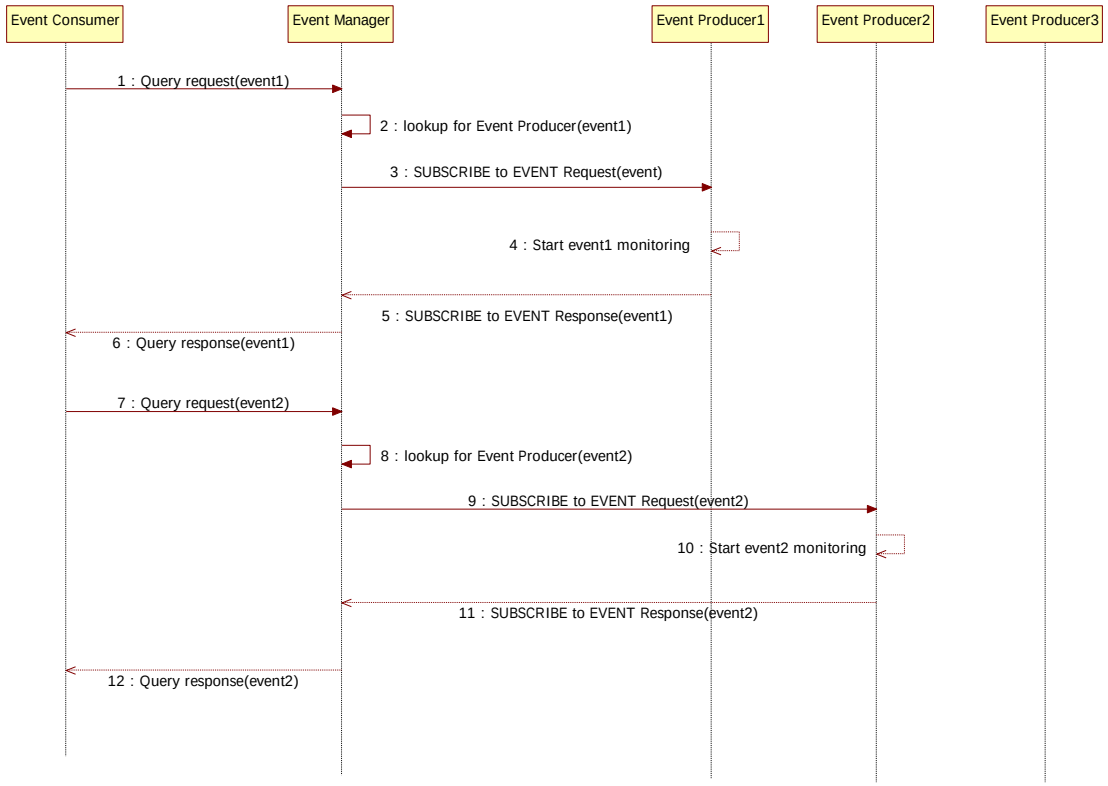


Figure 89: SUBSCRIBE to EVENT sequence diagram 2

4.6.11. NOTIFY EVENT message

This message let the customer know when a trigger condition has occurred in the Event Producer (event). The user must have sent the SUBSCRIBE to EVENT message previously in order to be associated to this event in the Event Manager. One EVENT NOTIFICATION message is sent per event generated.

Origin	**Event Producer**
Target	Event Manager
Transmission Mode	Unicast
Payload	Event-oriented services or events' name and the result of these events.

Chart 47: NOTIFY EVENT message

NOTIFY EVENT message is sent to the Event Manager when a trigger condition is generated in the device which is monitored by the Event Producer. The Event Producer is responsible for managing the event generated and sending the message.

For example, a person (Event Consumer) subscribes to the "notifyPresence" event-oriented service; if anyone enters to the customer's house without permission, this user of the service would receive a NOTIFY EVENT message in his / her Smartphone. If the customer had not

been registered in the Event Manager, this one never would receive the NOTIFY EVENT message.

The following sequence diagram shows an Event Customer subscribing to two event-oriented services provided the Event Producer 1 and Event Producer 2, respectively (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12). An unspecified time later an event on Event Producer 1 is triggered (13), so a NOTIFY EVENT message is sent to the Event Manager (14) that routes it to the Event Customer (15).

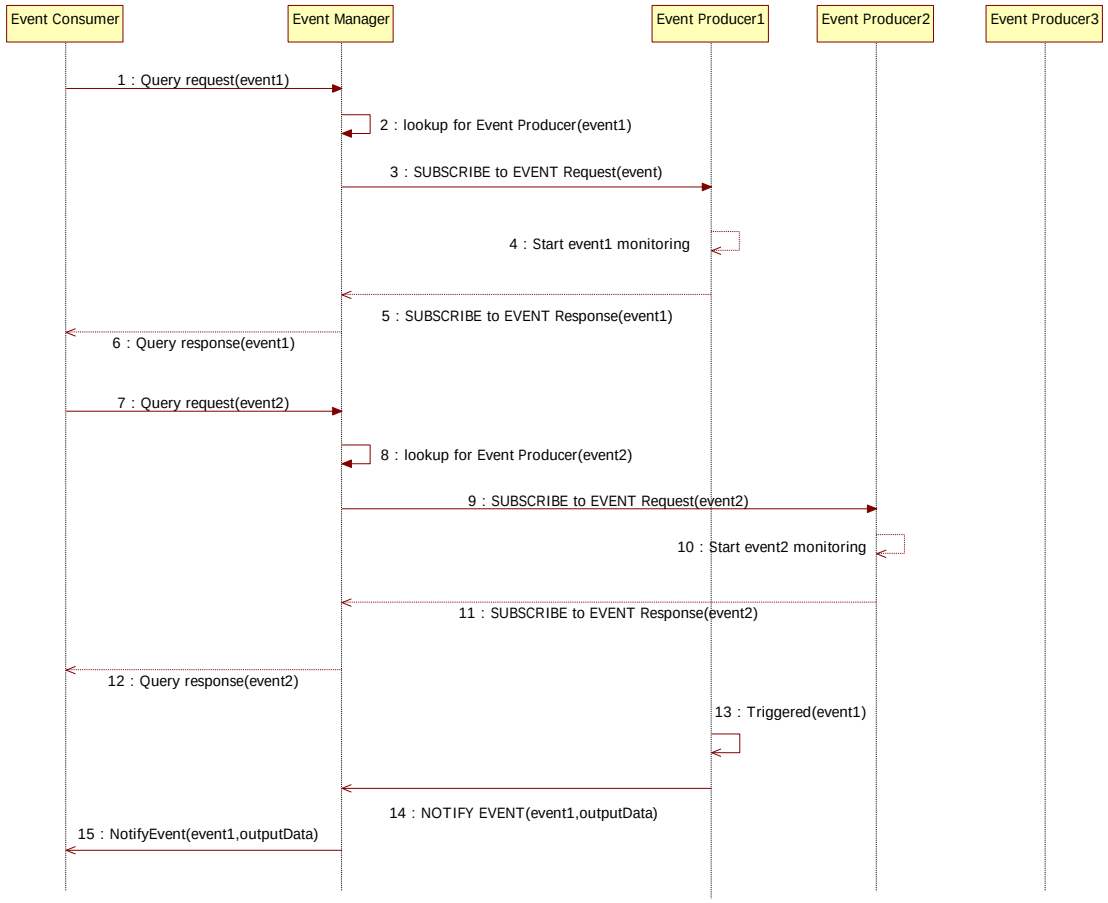


Figure 90: NOTIFY EVENT sequence diagram

4.6.12. Messages in JSON format

In constrained networks it is very important to save processing capabilities when exchanging requests and responses. Specification of these messages can be represented with XML, JSON or other. We have choose JSON because is lighter and easy-to-use compared to others. In the following sections we will describe in detail the previous messages in JSON format.

4.6.12.1. HELLO Request message

```
{
"transport": "j2me.radiogram",
"envelope": "JSON-2.0",
"target": "<IP or MAC address in broadcast>/<Event Managers and/or
Registries>",
“deviceID”: “<number>”,
“deviceInfo”:
        {
                “location”:
                {
                        “latitude”: <latitude>,
                        “longitude”: <longitude>,
                }
                “manufacturer”: <manufacturer>,
                “technicalFeatures”: [<technical features>],
                “extraInfo”: [<other information>],
}
"events":
        {
                "<nameEvent1>":
                {
                        "parameters": [{ "type": <type inputData1> },
                                       { "type": <type inputData2> }],
                        "returns":    [{ "type": <type outputData1> },
                                       { "type": <type outputData2> }]
                }
                "<nameEvent2>":
                {
                        "parameters": [{ "type": <type outputData1> }],
                        "returns":    [{ "type": <type outputData2> }]
                }
        }
"services":
        {
                "<nameService1>":
                {
                        "parameters": [{ "type": <type inputData1> },
                                       { "type": <type inputData2> }],
                        "returns":    [{ "type": <type outputData1> },
                                       { "type": <type outputData2> }]
                }
                "<nameService2>":
                {
                        "parameters": [{ "type": <type outputData1> }],
                        "returns":    [{ "type": <type outputData2> }]
                }
        }

}
```

Figure 91: HELLO Request message in JSON format

“transport”	The underlying protocol in which the message is encapsulated. In our case, within the WSAN would be Radiogram protocol.
“envelope”	The format of the transmitted message. In our case is JSON but it could be XML, for example.
“target”	The destination of the transmitted message and the software component which will handle it.
“origin”	The source of the transmitted message and the software component which will handle it.
“deviceID”	Number assigned to the device that makes easier lookup information for the Event Managers and / or Registries.
“deviceInfo”	Complementary data of the device which might be useful to make dashboards in order to view the most common device information and make reports about their performance, etc.

“events”	This field is labeled with the name of the service event-oriented is associated to the method of the agent which offers the services (e.g. “notifyPresence”). The agent’s method might require an array of input data’s types to accomplish the service (e.g. the event service “notifyPresence” might require the time when monitor, so input data would be start time and finish time (e.g. notifyPresence(“12.00”,”14.00”)). If the service does not require any input parameter, the input data would be specified with “void”. Input parameters are usually two or three. This method will return a specified data type (e.g. “notifyPresence” event returns a Boolean). This field usually carries with one output parameter.
“services”	This field is analogous to the previous “events”.

Chart 48: HELLO Request message JSON fields

4.6.12.2. HELLO Response message

```
{
   "transport": "j2me.radiogram",
   "envelope": "JSON-2.0",
   "target": "<IP or MAC address>/<Event Producer>",
   "origin": "<IP or MAC address>/<Event Producers and/or Registries>",
   "events": {"<nameEvent1>", "<nameEvent2>"}
   "services": {"<nameService1>", "<nameService2>"}
}
```

Figure 92: HELLO Response message in JSON format

“transport”	The underlying protocol in which the message is encapsulated. In our case, within the WSAN would be Radiogram protocol.
“envelope”	The format of the transmitted message. In our case is JSON but it could be XML, for example.
“target”	The destination of the transmitted message and the software component which will handle it.
“origin”	The source of the transmitted message and the software component which will handle it.
“events”	This field is labeled with the name of the service event-oriented is associated to the method of the agent which offers the services (e.g. “notifyPresence”).
“services”	This field is analogous to the previous “events”.

Chart 49: HELLO Response message JSON fields

4.6.12.3. PUBLISH Request message

```
{
       "transport": "j2me.radiogram",
       "envelope": "JSON-2.0",
       "target": "<IP or MAC address in broadcast>/<Event Managers and/or Registries>",
       "origin": "<IP or MAC address>/<Event Producer>",
      “deviceID”: <number>
       "events":
       {
              "<nameEvent1>":
              {
                     "parameters": [{ "type": <type inputData1> },
                                    { "type": <type inputData2> }],
                     "returns":    [{ "type": <type outputData1> },
                                    { "type": <type outputData2> }]
              }
              "<nameEvent2>":
              {
                     "parameters": [{ "type": <type outputData1> }],
                     "returns":    [{ "type": <type outputData2> }]
              }
       }
       "services":
       {
              "<nameService1>":
              {
                     "parameters": [{ "type": <type inputData1> },
                                    { "type": <type inputData2> }],
                     "returns":    [{ "type": <type outputData1> },
                                    { "type": <type outputData2> }]
              }
              "<nameService2>":
              {
                     "parameters": [{ "type": <type outputData1> }],
                     "returns":    [{ "type": <type outputData2> }]
              }
       }
}
```

Figure 93: PUBLISH Request message in JSON format

“transport”	**The underlying protocol in which the message is encapsulated. In our case, within the WSAN would be Radiogram protocol.**
“envelope”	The format of the transmitted message. In our case is JSON but it could be XML, for example.
“target”	The destination of the transmitted message and the software component which will handle it.
“origin”	The source of the transmitted message and the software component which will handle it.
“deviceID”	Number assigned to the device that makes easier lookup information for the Event Managers and / or Registries.
“events”	This field is labeled with the name of the service event-oriented is associated to the method of the agent which offers the services (e.g. “notifyPresence”). The agent’s method might require an array of input data’s types to accomplish the service (e.g. the event service “notifyPresence” might require the time when monitor, so input data would be start time and finish time (e.g. notifyPresence(“12.00”,”14.00”)). If the service does not require any input parameter, the input data would be specified with “void”. Input parameters are usually two or three. This method will return a specified data type (e.g. “notifyPresence” event returns a Boolean). This field usually carries with one output parameter.
“services”	This field is analogous to the previous “events”.

Chart 50: PUBLISH Request message JSON fields

4.6.12.4. PUBLISH Response message

The message is the same as HELLO Response message.

4.6.11.5. SUBSCRIBE to EVENT Request message

```
{
      "transport": "j2me.radiogram",
      "envelope": "JSON-2.0",
      "target": "<IP or MAC address>/<Event Producer>",
      "origin": "<IP or MAC address>/<Event Manager>",
      "event": "<nameEvent>"
}
```

Figure 94: SUBSCRIBE to EVENT message in JSON format

"transport"	**The underlying protocol in which the message is encapsulated. In our case, within the WSAN would be Radiogram protocol.**
"envelope"	The format of the transmitted message. In our case is JSON but it could be XML, for example.
"target"	The destination of the transmitted message and the software component which will handle it.
"origin"	The source of the transmitted message and the software component which will handle it.
"events"	This field is labeled with the name of the service event-oriented is associated to the method of the agent which offers the services (e.g. "notifyPresence").

Chart 51: SUBSCRIBE to EVENT Request message JSON fields

4.6.12.6. SUBSCRIBE to EVENT Response message

```
{
      "transport": "j2me.radiogram",
      "envelope": "JSON-2.0",
      "target": "<IP or MAC address>/<Event Manager>",
      "origin": "<IP or MAC address>/<Event Producer>",
      "event": "<nameEvent>"
}
```

Figure 95: SUBSCRIBE to EVENT Response message in JSON format

The fields' descriptions are the same as SUBSCRIBE to EVENT Request message described above.

4.6.12.7. NOTIFY EVENT message

```
{
      "transport": "j2me.radiogram",
      "envelope": "JSON-2.0",
      "target": "<IP or MAC address>/<Event Manager>",
      "origin": "<IP or MAC address>/<Event Producer>",
      "event": "<nameEvent>"
      "returns": "<outputData>"
}
```

Figure 96: NOTIFY EVENT message in JSON format

“transport”	**The underlying protocol in which the message is encapsulated. In our case, within the WSAN would be Radiogram protocol.**
“envelope”	The format of the transmitted message. In our case is JSON but it could be XML, for example.
“target”	The destination of the transmitted message and the software component which will handle it.
“origin”	The source of the transmitted message and the software component which will handle it.
“event”	This field is labeled with the name of the service event-oriented. In the implementation is associated to the method of the agent which offers the event-oriented service (e.g. “notifyPresence”).
“returns”	The data which triggered the event.

Chart 52: NOTIFY to EVENT message JSON fields

CHAPTER 5: SYSTEM ANALYSIS

In addition to the nSOM architecture specification, we have validated with effectiveness its performance combining simulations and experimental measures. The validated infrastructure has been deployed in the CITSEM building (Centro de Investigación en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad). This chapter introduces a description of the scenario deployment, the main use cases, and the experimental measures that have been taken.

5.1. Deployment scenario

The scenario consists of a WSAN following the star topology which is interconnected to the IoT gateway. The scenario is divided in three parts; the WSAN, the IoT gateway, and the external system (Internet) from which requests are received.

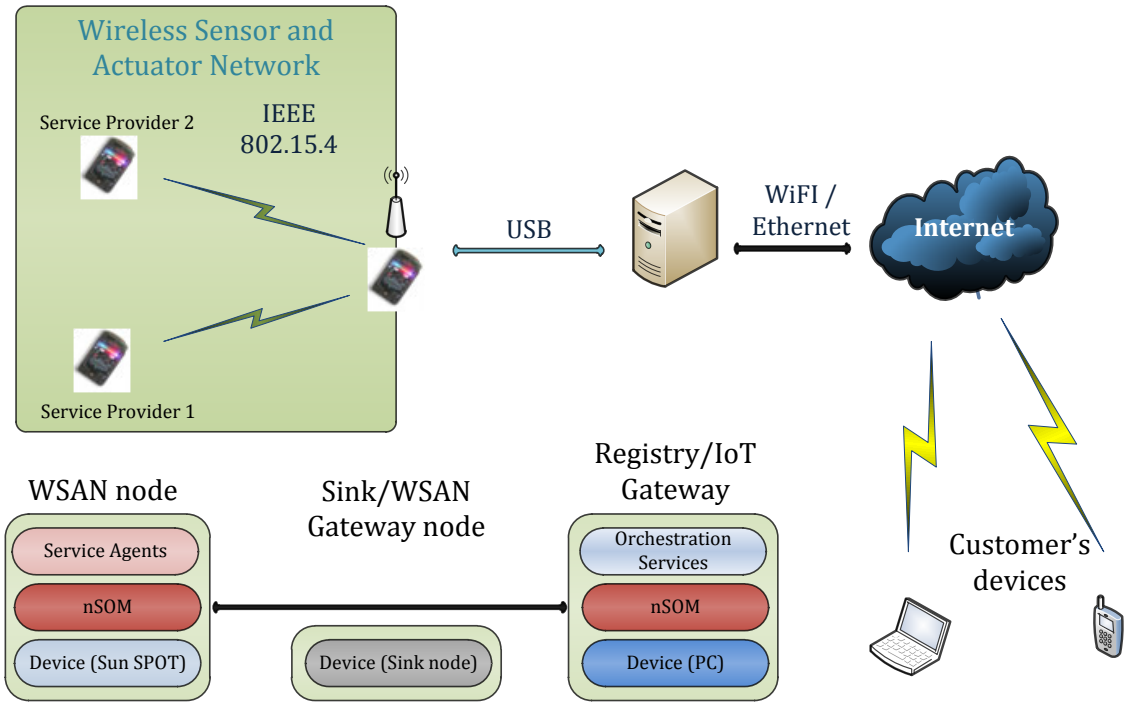


Figure 97: Network scheme of the deployment scenario

nSOM architecture is embedded in every node of the WSAN. The WSAN nodes plays different roles depending on the services that they provide (e.g. Temperature service). Services are implemented by applications called "agents" deployed over nSOM (e.g. Temperature agent will provide temperature service). WSAN nodes are physically connected to the IoT gateway through a sink (also called WSAN node) which is connected via USB to the IoT gateway. The

sink does not embed the nSOM architecture, but its function is limited to pass the data from / to the WSAN at a physical layer. Agents are the Service Providers.

The IoT gateway is a computer that plays a double role in the scenario since it is the interface between Service Providers and Customers. From WSAN point of view it will be the Registry of services. To external systems such as Internet it will be understood as a gateway to a WSAN; it "virtualizes" the services offered by the WSAN with orchestration services techniques (Lucas Martínez, 2013). IoT gateway is able to communicate directly with each node via nSOM messages because it embeds the architecture in the same manner as WSAN nodes do.

The IoT Gateway implements a REST-based API that allows external systems invoking the WSAN services, and it also allows performing remote actions in the Registry. Registry is "virtualized" as a resource called "wooreg". The table below summarizes the main commands to interact with the IoT gateway.

Registry

Action performed	Method	Endpoint
Create new service or device entry in the Registry	POST	/ wooreg
List all devices registered with their associated services	GET	/ wooreg
Retrieve services provided by a specific device	GET	/wooreg?deviceID={deviceID}
List devices classified with a specific type and their associated services.	GET	/wooreg?deviceType={type}
Delete all data associated to a specific device	DELETE	wooreg/{deviceID}

Chart 53: Actions performed to Regisry with REST API-based

Services

Action performed	Method	Endpoint
Service request to a specific device	GET	*/{deviceID}/{servicio}*

Chart 54: Actions performed to services with REST API-based

Finally, customers are those people who request services with a RESTful device, that is to say, any device with a web browser and internet connection (e.g. a laptop, smart phone, desktop computer, etc.). The services requested are provided by agents of the WSAN; however, customers do not have direct connection with them. What is more, they do not realize of the underlying application messages.

Next figures display the equipment underlying the physical infrastructure deployed. On the left figure two Sun SPOT motes (the ones that light) with the Sun SPOT sink have been used to perform the WSAN. On the right figure, an ASUS computer is used for the IoT gateway.

Figure 98: Equipment used in the deployment scenario

5.2. Use cases of the scenario

Within our system infrastructure it can be differentiated two use cases. The first corresponds to "Service exposure" and it is a mandatory use case to accomplish the second one, "Service request". Interacting with the system are three roles or actors; the "Customer", the "Service Provider", and the "Registry".

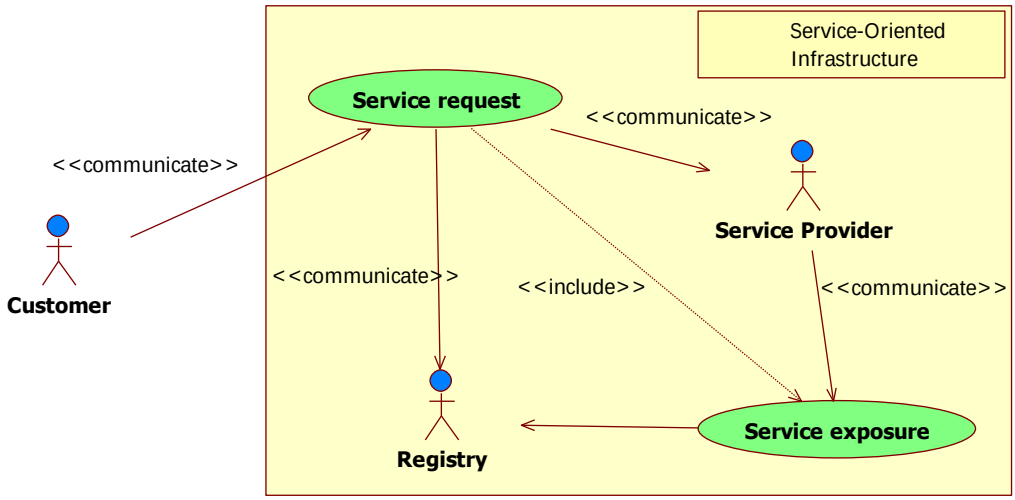


Figure 99: Use case diagram for the deployment scenario

The Customer is a person who request for a service to the Registry via RESTful device with internet connection. This person can request for a service at any time and he / she will receive the response in real time.

The Service Providers are the agents deployed on the nSOM architecture embedded in the physical nodes of the WSAN. When the nodes are reseated, the agents will send their services with the HELLO message by broadcast.

Registry stores the services, the associated agents and the physical direction of the WSAN node in which they are deployed.

5.2.1. Service exposure

The Service Provider and the Registry are the actors involved in this use case. This use case must be occurred before the Service request use case in order to report the services to the Registry. This use case is only once performed with the setup network.

The communication is initiated by the powering up nodes of the WSAN which have deployed the agents that send the HELLO message by broadcast mode. Once the register have the information about the services, their agents and the physical direction of the nodes, then they are available to any Customer can request for them at any time.

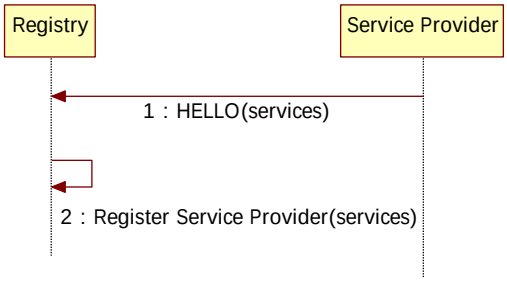


Figure 100: Service exposure sequence diagram

5.2.2. Service request

This use case makes the interaction between the Customer and the Registry, and in the other hand, the Registry with the Service Provider.

The Customer starts the communication at any time with a request service to the Registry (1). Then, the Registry lookup (2) for the Service Provider of that service and it will send the SEND DATA REQUEST nSOM message (3).

Note how the Registry acts as an IoT gateway; the application communication between the Registry and the Customer is via HTTP (1), (7), and the application communication between the Registry and the Service Provider is via nSOM protocol (3), (5).

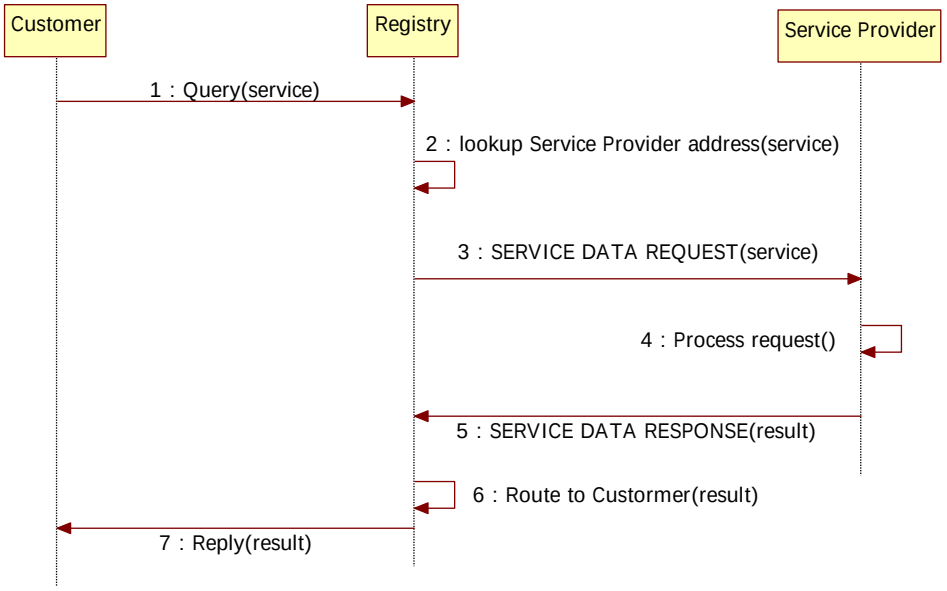


Figure 101: Service request sequence diagram

5.3. Testing nSOM architecture

We will take some measures to evaluate the performance of nSOM architecture embedded in a Sun SPOT mote. Specifically, the three phases that happen when the node starts are analyzed (for further details see figure 72); components instantiation, components deployment, and the starp-up execution of these components. The components analyzed are the DIAL, the Service Manager, the Service Execution Platform and the agents. These tests are linked to the Service exposure use case.

The next figure shows a summary of the real average time of each phase. It also exposes the sequence order of their activation in the Sun SPOT mote.

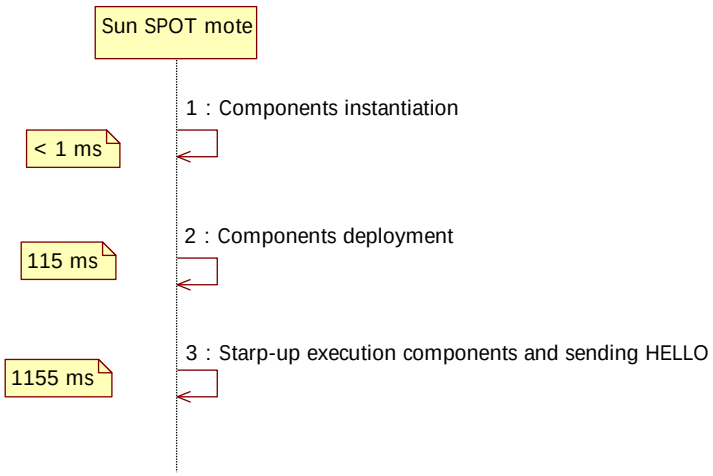


Figure 102: nSOM architecture start-up phases

With regard to the Service request use case, we have measured the request processing time.

In the next sections will describe the mentioned tests realized in detail. Furthermore, the last section presents in a table the value of each measure.

The organization of testing is as follows; first, we will measure the node reset. This test is the more general and compresses the three phases represented in the figure above. The next tests are particular cases of the first one, and are related to each phase. The last test is reserved to the request processing time. Every test will consist of 50 trials, half in a real Sun SPOT mote, and the other half in a simulated Sun SPOT mote. Sun SPOT offer the Solarium emulator, a tool that allows simulating wireless sensor and actuator networks with Sun SPOT motes.

5.3.1. Sun SPOT mote reset

The Sun SPOT mote takes a time in starting a being fully operative. This covers the three phases that nSOM requires to send the HELLO message; components instantiation, components deployment, and the starp-up execution of these components.

The next graph shows a comparative between the measures with a real mote and a simulated one.

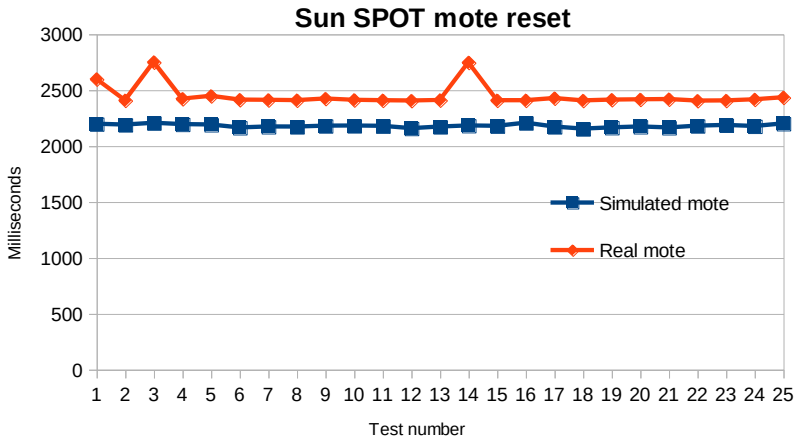


Figure 103: Sun SPOT mote reset test graph

	Average	Mode	Maximum	Minimum
Real	2448.68	2408 (4 times)	2750	2404
Simulated	2179.68	2172 (2 times)	2207	2154

Chart 55: Sun SPOT mote reset statistical values

Comparing the real and simulated measures, the results are the expected. nSOM architecture embedded in a simulated mote powers up before than a real one. With regard to the real mote

results, the line are quasi linear, except two random trials. So, we can conclude that the nSOM architecture deployment in a Sun SPOT mote is very stable.

5.3.2. Components instantiation

Components instantiation is the phase with less time consumption. Most of the 50 trials performed per each component take less than one millisecond to be instantiated. Therefore, regarding time consumption, we can consider this phase despicable.

5.3.3. Components deployment

As the before test, components deployment do not added notably delays to the start up node. However, as it can be observed in the graph below, Service Manager component is the exception with an average of 115.44 milliseconds in a real mote.

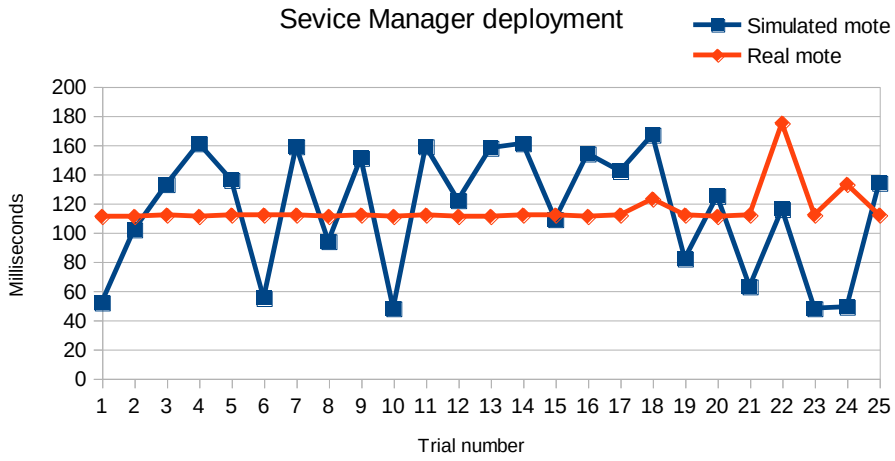


Figure 104: Service Manager component deployment test graph

	Average	Mode	Maximum	Minimum
Real	115.44	112 (12 times)	175	111
Simulated	115.2	159 (2 times)	167	52

Chart 56: Service Manager deployment statistical values

Regarding real and simulated mote, the measures are not the expected because the real one has a better performance than the simulated one. Furthermore, the non linear graph indicates that it is not stable. A hypothetical explanation for this rarity is that the Solarium tool does not characterize accurately the mote when multiple chains of software invocations are produced on it. The Service Manager component is responsible of composite the HELLO message in JSON format, so it requires invoking several components and vice versa. Therefore is one of the components that consume more software resources of the mote.

5.3.4. Starp-up execution components and sending HELLO

This test corresponds to the last phase when the mote powers up. It encompasses the time from the launching sequentially each component deployed in the nSOMContainer up to sending the HELLO message.

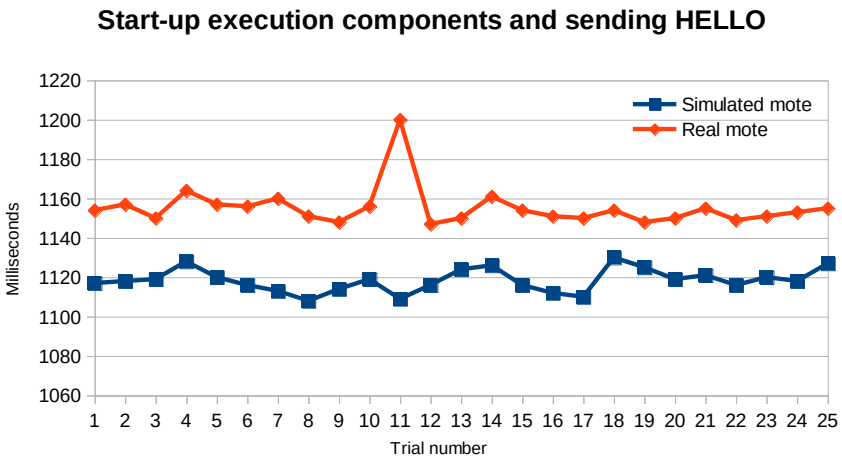


Figure 105: Start-up execution components and sending HELLO test graph

	Average	Mode	Maximum	Minimum
Real	1155.24	1150 (4 times)	1200	1147
Simulated	1118.44	1116 (4 times)	1130	1108

Chart 57: Starp-up execution components statistical values

Although both real and simulated measures are coherent, it can be observed certain instability in both of them. This can happen due to the composition of the HELLO message in JSON format and its encapsulation in a physical layer datagram. So, in addition of multiple chain software invocations, hardware resources such as the antenna are required. This phase involves both software and hardware operations, so it could be coherent to think about a little more disparity among their results than the mentioned tests.

5.3.5. Request processing time

This test is associated to the Service request use case and it measures the time than the mote takes to process a request. The time goes from the instant that the request datagram is received from the Registry to the instant that the response datagram is returned to the Registry again.

In the next graph shows that the simulated mote is about six times faster in processing the request. On the other hand, it can be pointed out the stability of the mote when processing requests.

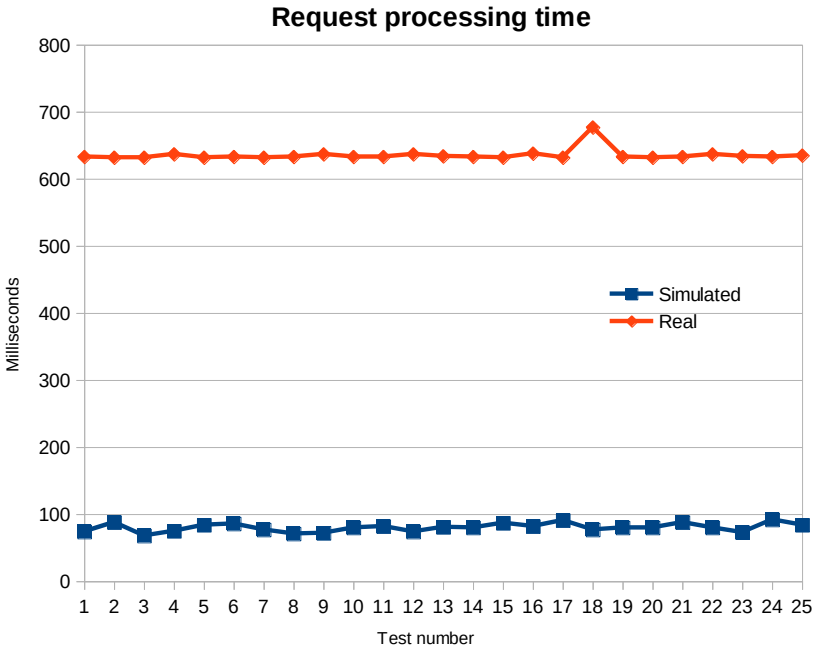


Figure 106: Request processing time test graph

	Average	Mode	Maximum	Minimum
Real	635.48	633 (9 times)	677	632
Simulated	80.24	80 (5 times)	92	71

Chart 58: Request processing time statistical values

5.3.6. Table of measures

This section exposes a table with detailed values of the taken measures in a Sun SPOT mote with the nSOM architecture embedded. They have been used in the previous graphs of tests.

Trial number	Sun SPOT mote reset (real)	Sun SPOT mote reset (simulated)	Request processing time (real)	Request processing time (simulated)	Start-up execution components (real)	Start-up execution components (simulated)
1	2598	2197	633	74	1154	1117
2	2408	2189	632	88	1157	1118
3	2750	2206	632	68	1150	1119
4	2422	2194	637	75	1164	1128
5	2447	2190	632	84	1157	1120
6	2413	2164	633	86	1156	1116
7	2411	2173	632	77	1160	1113
8	2409	2172	633	71	1151	1108
9	2423	2180	637	72	1148	1114
10	2411	2181	633	80	1156	1119
11	2408	2178	633	82	1200	1109
12	2405	2157	637	74	1147	1116
13	2410	2172	634	81	1150	1124
14	2746	2182	633	80	1161	1126
15	2408	2178	632	87	1154	1116
16	2408	2207	638	82	1151	1112
17	2427	2172	632	91	1150	1110
18	2406	2154	677	77	1154	1130

19	2413	2165	633	80	1148	1125
20	2416	2173	632	80	1150	1119
21	2418	2164	633	88	1155	1121
22	2404	2180	637	80	1149	1116
23	2406	2187	634	73	1151	1120
24	2416	2177	633	92	1153	1118
25	2434	2200	635	84	1155	1127

Chart 59: Test measured values

Trial number	Service manager deployment (real)	Service manager deployment (simulated)
1	2598	52
2	2408	102
3	2750	133
4	2422	161
5	2447	136
6	2413	55
7	2411	159
8	2409	94
9	2423	151
10	2411	48
11	2408	159
12	2405	122
13	2410	158
14	2746	161
15	2408	109
16	2408	154
17	2427	142
18	2406	167
19	2413	82
20	2416	125
21	2418	63
22	2404	116
23	2406	48
24	2416	49
25	2434	134

Chart 60: Service Manager component deployment measured values

PART III: CONCLUSIONS AND FUTURE WORKS

CONCLUSIONS

FUTURE WORKS

CONCLUSIONS

The contents of this Final Project Dissertation are divided into two parts.

The first part presents the Internet of Things' technologies with a double perspective: on the one hand, the "open source" hardware platforms. They can be seen as "things" which collect data from their environment. On the other hand, cloud platforms for mass storage of things' data are analyzed.

The second part focuses on a particular case study of the Internet of Things; the specification of an architecture software for a Wireless Sensor and Actuator Network (WSAN) that allows external systems accessing their services.

The approach to the analysis of the emerging technologies of the Internet of things, especially of the clouds, has been a constant review of them. New technologies appear continuously and in a short space of time. In addition, the novelty of them implies a diffuse documentation and little solid data, as well as a lack of references.

I will establish a classification of the conclusions in accordance with the structure of this Final Project Dissertation; "open source" hardware platforms, cloud platforms and the case study.

Open hardware platforms

There are countless platforms designed for a specific purpose in the market. In contrast, this Project has focused on "incomplete" hardware platforms that allow you to develop solutions tailored to the needs not covered by mercantilist general solutions. In addition, platforms follow the "open source" philosophy. Therefore, the firmware or operating system embedded will be "open source" too. Firmware and operating systems open source give more versatility to mount nSOM.

Although the market share[1] of the Internet of things does not exceed 1%, during the realization of the project, I have seen that there is a wide range of open source platforms that suggest applications for the Internet of things. The vast majority of these platforms derived from Arduino family, one of the most popular. In fact, one of the studied platforms, Waspmote derived from Arduino.

1 Gartner Survey "Hype Cycle for Smart City Technologies and Solutions 2012"

The huge amount of multiple devices makes arduous the task of classification of them. However, I found interesting to organize platforms analyzed depending on the item that will characterize its capacity of processing: the processor. In this regard, I have differentiated platforms based on microcontroller and microprocessor-based.

Processing characteristics of microprocessor-based platforms are more powerful. The studied ones embed operating systems similar to desktop operating systems. On the contrary, based microcontroller platforms have less processing power, and they usually incorporate a set of libraries for the management of low peripherals. However, microcontroller-based platforms are righter for harvesting data of their environment.

I have observed that there is a tendency in this type of platforms to provide a physical infrastructure with various types of connectors and interfaces to facilitate interconnection with other systems or modules. An example of this is Waspmote PRO which is oriented to wireless sensor networks and allows plugging a collection of modules and a wide range of sensors.

Finally mentioning the upward trend of platforms based on microcontroller that carry embedded TCP/IP – UDP/IP socket that allows direct connection to the Internet.

Cloud platforms

Undoubtedly, clouds offer a wide range of possibilities for developing applications on the web with data sending devices, for example the studied ones. In addition, they present the advantage of the availability data from anywhere in the world with an Internet connection.

Clouds appear as a solution for the mass storage and the processing of the enormous quantity of traffic or “big data” from 100 000 million devices foreseen for the year 2020. In addition to the storage, many of them provide services added as a species of gangplank to web services of third parties as Twitter, Facebook, Dropbox, Prowl or Twilio

With the exception of Xively, the rest of analyzed platforms are in a phase of "gestation" and the information provided is diffuse. However, clearly notes that all offer access to their resources through a REST API, and the favorite application-level data transfer format is JSON. More mature clouds as Xively offer more options such as XML, CSV or PNG.

Furthermore, they all offer a free account for the developer with a graphical web interface to manage the devices and to create applications from the extracted information.

nSOM

The main purpose of nSOM is enabling the development of applications regardless of the hardware platform used. Reusable software architecture facilitates agile application

development and also prevents any software development have to start from scratch, which means save time and increase the possibilities for innovation.

During the phase of design and implementation, nSOM software components have been organized for the incorporation of the new ones such as the Device Manager, as well as the specification of messages to nSOM evolves to the SOA 2.0 philosophy.

Analysis of hardware platforms has been very useful to better understand broader connectors and interfaces and thus make a more accurate design of the Device Manager component. This component focuses on obtaining device information such as resources hardware and geographic location.

Another aspect to highlight is the manner in which devices talk to each other. Therefore, it is the vital importance to clearly specify messages, information and the "moment" in which they will be sent. In this project the formal specification of messages for events has been addressed which gives more flexibility to inter-node communications.

FUTURE WORKS

The combination of the nSOM architecture along with the studied technologies development would lead to an endless number of applications. I highlight the following ones:

- Taking advantage of the low-cost platforms such as Arduino, Netduino or BeagleBone hardware, they could be applied for the nSOM architecture deployment. In addition, the Device Manager component could be used to develop applications in which each device send data periodically to a master node. This master node would have a database that could store data such as hardware resources, the location of the device, battery level, and other information that is considered to be interest. These data could be used to carry out studies and historical, and would facilitate control over a network with a large number of devices.
- The new specified messages could be used to carry information of the device to the node master when you have an event. For instance, to inform the above-mentioned application that mote runs out of battery.
- nSOM was initially coined for deployment in wireless sensor networks. Its functionality could be extended so that it exercised gateway between a network of heterogeneous devices and one cloud in which to store the data which could be available from anywhere in the world in any time. It could be interesting to connect with social media such as Twitter or Facebook to make applications of all kinds.

ANNEXES

ANNEX A: **Open hardware platforms' features comparison**

ANNEX B: **Blinking a LED program**

ANNEX A: Open hardware platforms' features comparison

A.1. Single-board microcontrollers

A.1.1. Official Arduino family

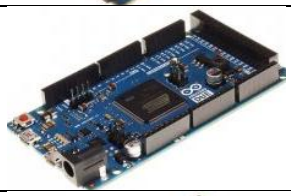
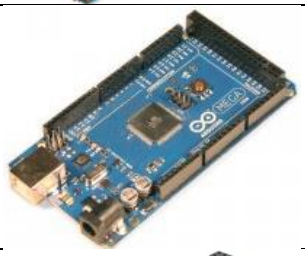
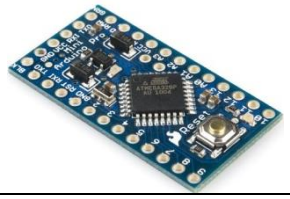
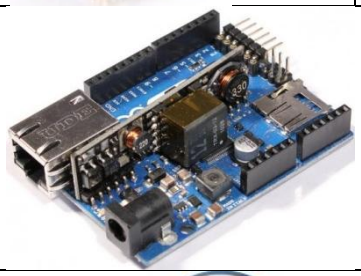
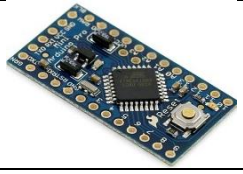
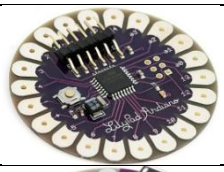
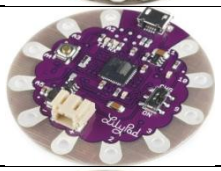
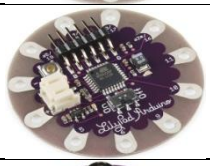
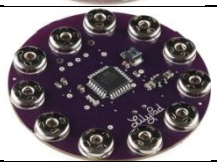
Name	Photo	Microcontroller	CPU speed	Program + data memory
UNO		ATmega328 (8-bit)	16 MHz	32 KB (Flash) + 2 KB SRAM
Leonardo		ATmega32u4 (8-bit)	16 MHz	32 KB (Flash) + 2.5 KB SRAM
DUE		AT91SAM3X8E (32-bit)	84 MHz	512 KB (Flash) + 96 KB SRAM
Mega 2560		ATmega2560 (8-bit)	16 MHz	256 KB (Flash) + 8 KB SRAM
Mega ADK		ATmega2560 (8-bit)	16 MHz	256 KB (Flash) + 8 KB SRAM
Micro		ATmega32u4 (8-bit)	16 MHz	32 KB (Flash) + 2.5 KB SRAM
Mini		ATmega328 (8-bit)	16 MHz	32 KB (Flash) + 2 KB SRAM

Chart 61: Arduino family features' comparison

Arduino (continuation)

Nano		ATmega168 (8-bit) ATmega328	16 MHz	16/32 KB (Flash) + 1/2 KB SRAM
Ethernet		ATmega328 (8-bit)	16 MHz	32 KB (Flash) + 2 KB SRAM
Esplora		ATmega32u4 (8-bit)	16 MHz	32 KB (Flash) + 2.5 KB SRAM
Bluetooth		ATmega328 (8-bit)	16 MHz	32 KB (Flash) + 2 KB SRAM
Fio		ATmega328P (8-bit)	8 MHz	32 KB (Flash) + 2 KB SRAM
Pro (328)		ATmega328 (8-bit)	16 MHz	32 KB (Flash) + 2 KB SRAM
Pro Mini (168)		ATmega168 (8-bit)	8 MHz	16 KB (Flash) + 1 KB SRAM

Arduino (continuation)

Name	Photo	Microcontroller	CPU speed	Program + data memory
LilyPad		ATmega168V (8-bit)	8 MHz	16 KB (Flash) + 1 KB SRAM
LilyPad USB		ATmega32u4 (8-bit)	8 MHz	32 KB (Flash) + 2 KB SRAM
LilyPad Simple		ATmega328 (8-bit)	8 MHz	32 KB (Flash) + 2 KB SRAM
LilyPad SimpleSnap		ATmega328 (8-bit)	8 MHz	32 KB (Flash) + 2 KB SRAM

A.1.2. Netduino family

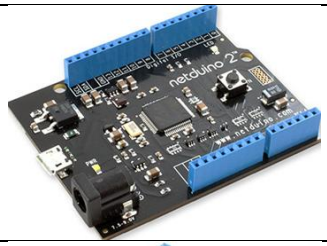
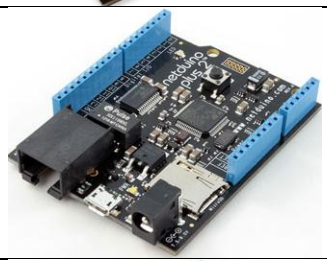
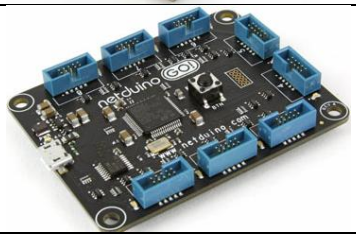
Name	Photo	Microcontroller	CPU speed	Memory
Netduino 2		STMicro STM32F2 (32-bit)	120 MHz	192 KB + 60 KB RAM
Netduino Plus 2		STMicroSTM32F4(32-bit)	168 MHz	384 KB + 100 KB RAM
Netduino GO!		STMicro STM32F4 (32-bit)	168 MHz	384 KB + 100 KB RAM

Chart 62: Netduino family features' comparison

A.1.3. Wireless sensor motes

Name	Photo	Microcontroller	CPU speed	Program + data memory
Sun SPOT		AT91SAM9G20 (32-bit)	400 MHz	8 MB (Flash) + 1 MB RAM
Waspmote PRO		Atmel ATmega 1281 (8-bit)	14 MHz	128 KB (Flash) + 8 KB SRAM
PowWow		MSP430F1612 (16-bit)	8 MHz	55kB (Flash) + 5kB RAM

Chart 63: Wireless sensor motes features' comparison

A.2. Single-board computers

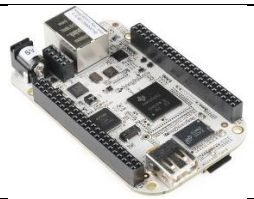
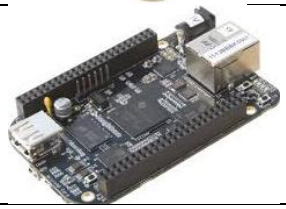
Name	Photo	Processor	SRAM	Connectors
Raspberry Pi		700 MHz ARM11 ARM1176JZF-S core	256 MB (Model A) 512 MB (Model B)	Analog audio output, analog video output, USB port, Ethernet, HDMI, CSI, DSI, microSD
BeagleBone		AM3359 500MHZ-USB Powered 720MHZ-DC Powered	256MB DDR2 400MHZ (128MB Optional)	USB, Ethernet
BeagleBone Black		Sitara AM3359AZCZ100 1GHz, 2000 MIPS	512MB DDR3L 800MHZ	USB, Ethernet, microSD, microHDMI

Chart 64: Single-board-computers features' comparison

Single-board computers (continuation)

Name	Photo	Processor	SRAM	Connectors
BeagleBoard		OMAP3530DCBB72 720MHz	Micron 2 GB NAND (256 MB) 4 GB C5 2 GB MDDR SDRAM (256 MB)	DVI-D, S-Video, Stereo out, stereo in
BeagleBoard-Xm		Texas Instruments Cortex A8 1GHz processor	Micron 4Gb MDDR SDRAM (512MB) 200MHz	DVI-D, S-Video, Stereo out, stereo in, LCD, 2 USB, Ethernet, RS-232, camera header

ANNEX B: Blinking a LED program

This annex aims to give an overview of the basic program "Blinking a LED" for each studied platform in order to compare with each other. For these hardware platforms the "Blinking a LED" program is analogous to "Hello, world" in programming languages.

B.1. Arduino

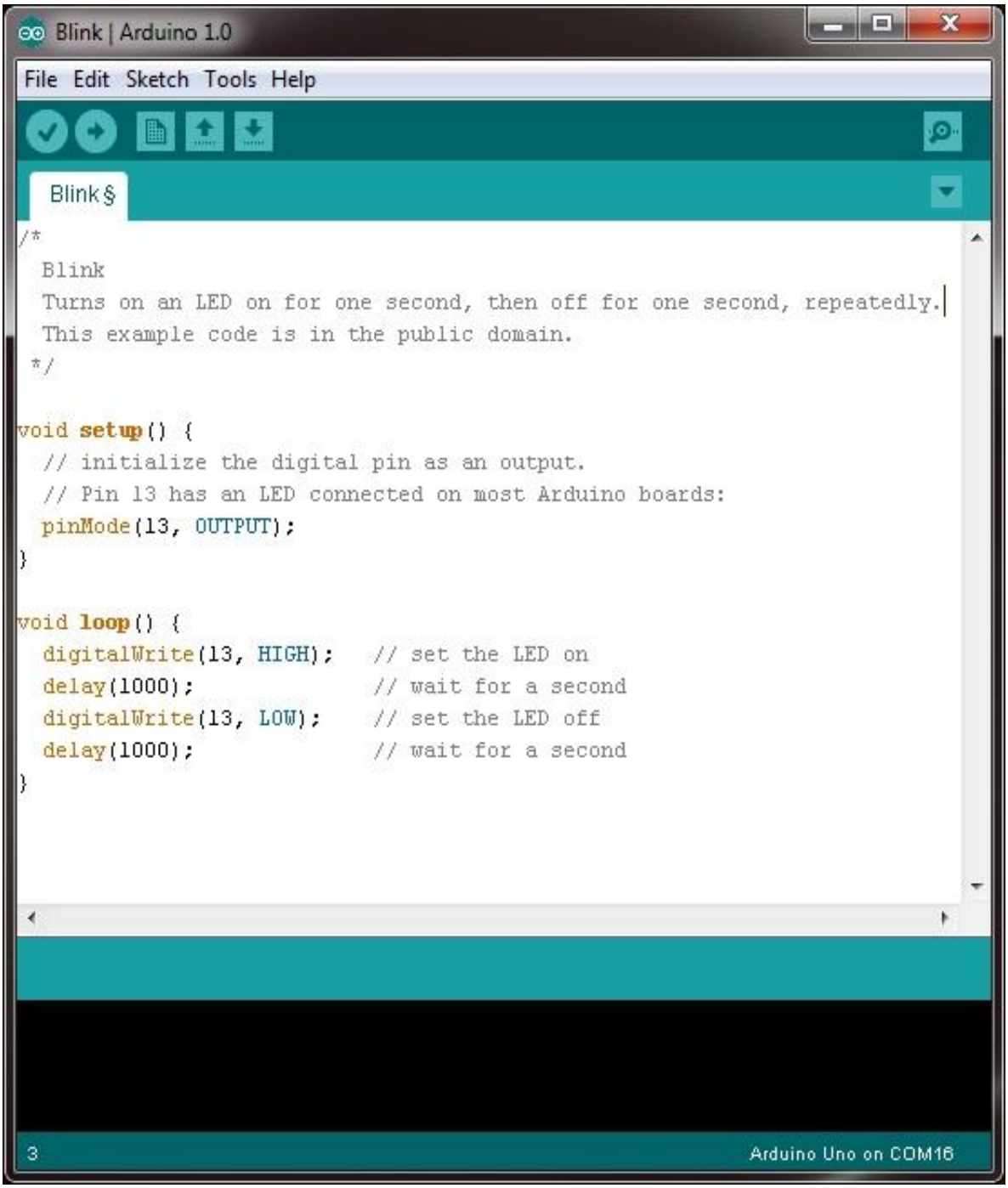


Figure 107: Blinking LED program for Arduino (RobotShop, 2013)

B.2. Netduino

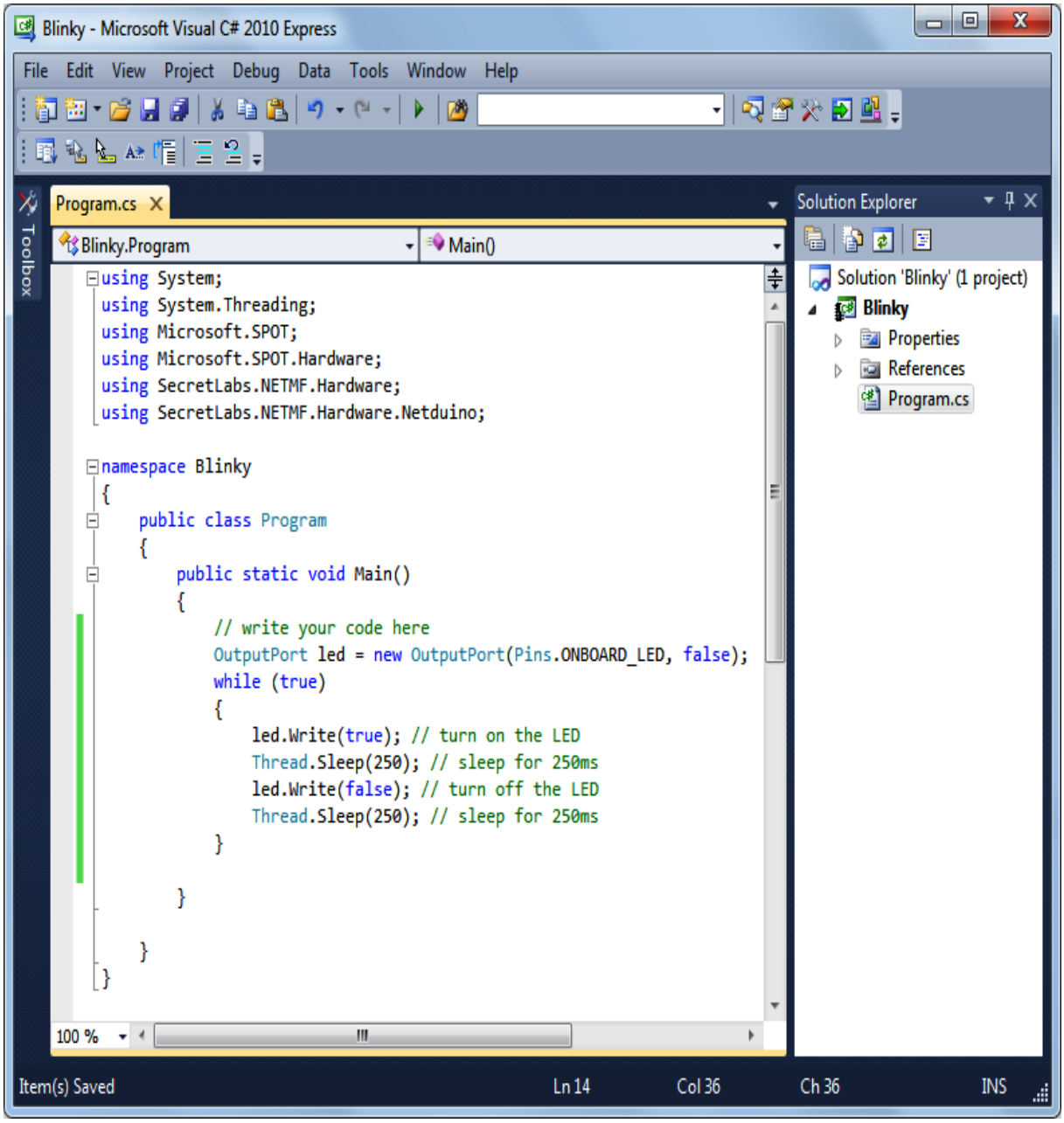


Figure 108: Blinking LED program for Netduino (Walker, 2012)

B.3. Sun SPOT

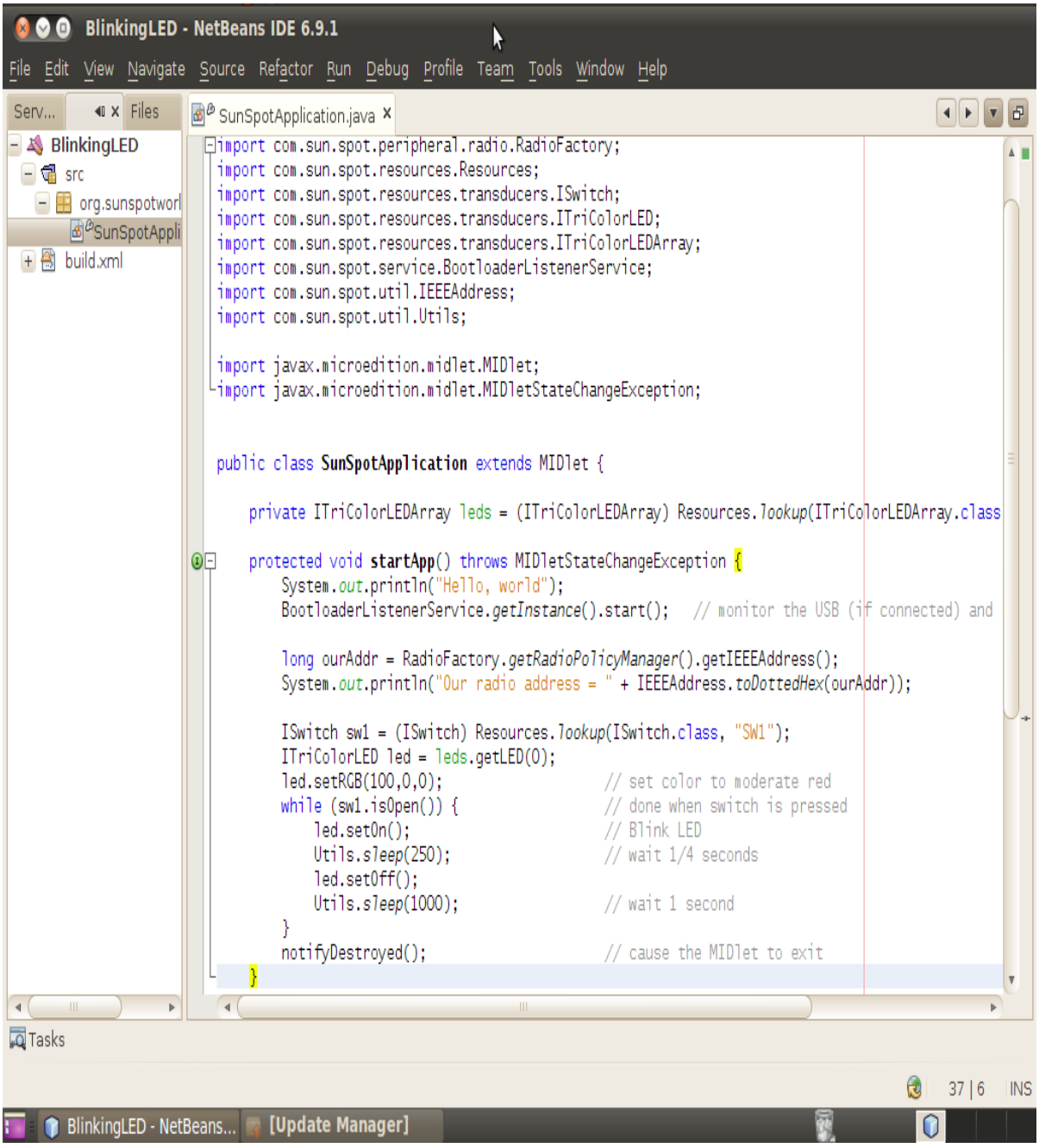


Figure 109: Blinking LED program for Sun SPOT

B.4. Waspmote PRO

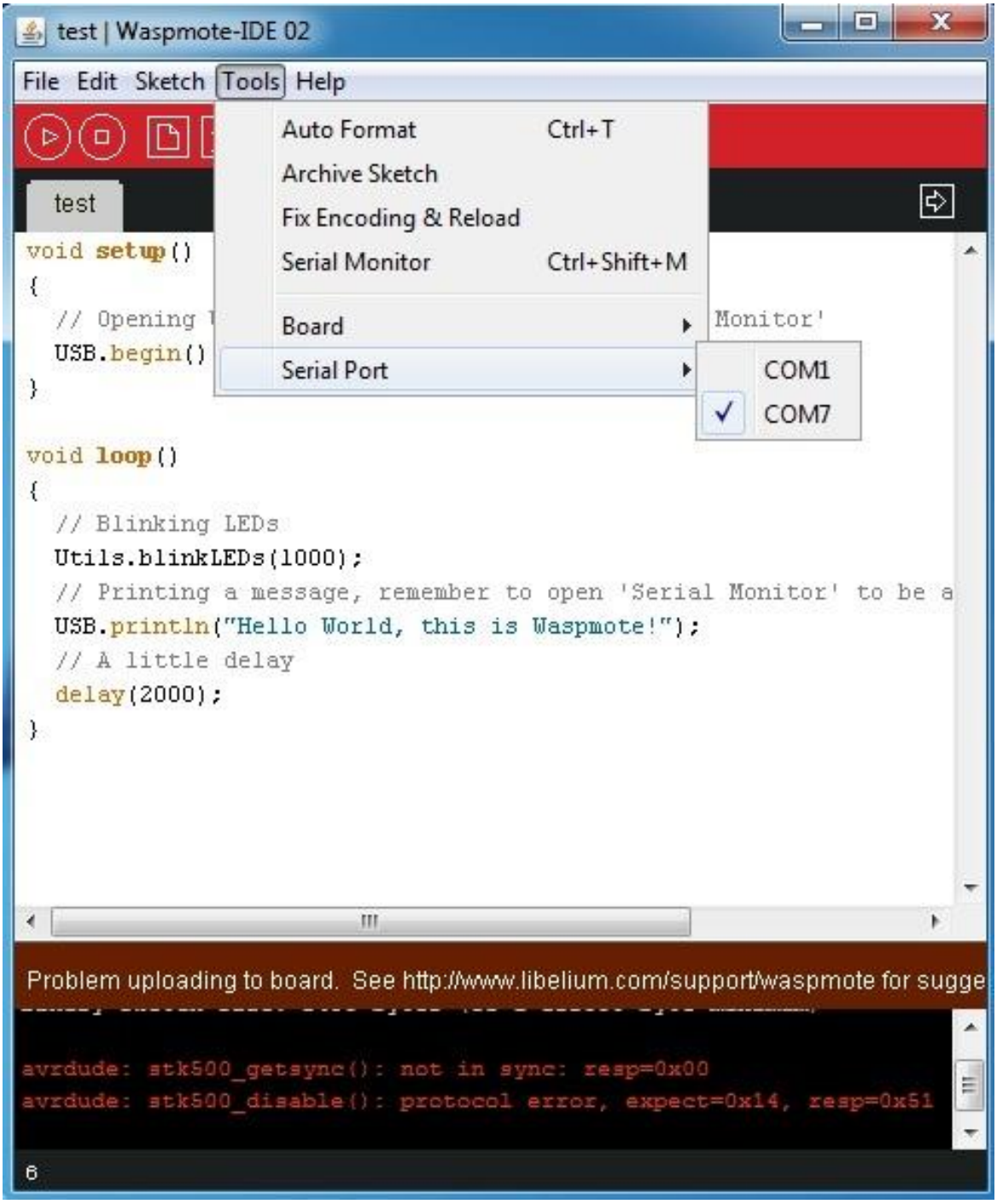


Figure 110: Blinking LED program for Waspmote PRO (Lib131)

B.5. Raspberry Pi

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(25, GPIO.OUT)

while True:
   GPIO.output(25, GPIO.HIGH)
   time.sleep(1)
   GPIO.output(25, GPIO.LOW)
   time.sleep(1)
```

Figure 111: Blinking LED program for Raspberry Pi (Richardson, et al., 2012)

B.6. BeagleBone

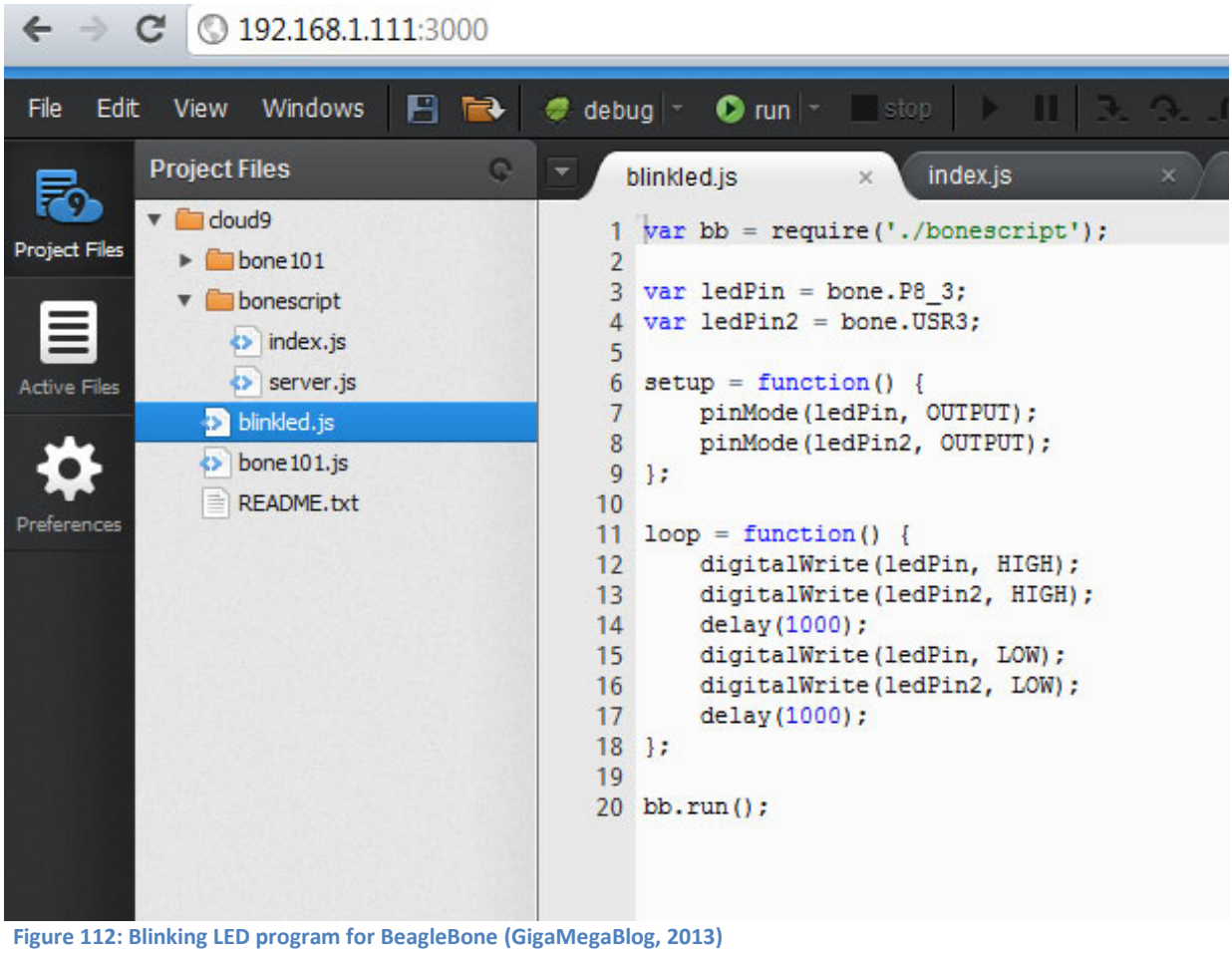


Figure 112: Blinking LED program for BeagleBone (GigaMegaBlog, 2013)

BIBLIOGRAPHIC REFERENCES

ANRG. 2013. Autonomous Networks Research Group. *Autonomous Networks Research Group.* [Online] Autonomous Networks Research Group, August 2013. http://anrg.usc.edu/ee579_2012/Group07/.

Arduino. 2013. Arduino. *Arduino.* [Online] Arduino, August 2013. http://www.arduino.cc/.

ArduinoBrasil. 2013. Arduino Brasil. *Arduino Brasil.* [Online] Arduino Brasil, August 2013. http://www.arduinobrasil.com/2011/01/uno/.

Axeda. 2013. Axeda. *Axeda.* [Online] Axeda Corporation, August 2013. http://www.axeda.com/.

Banzi, Massimo. 2008. *Getting Started with Arduino.* s.l. : Make, 2008. 978-0596155513.

Barragán, Hernando. 2013. Wiring. [Online] August 2013. http://wiring.org.co/. 2011-8376.

BeagleBoard. 2013. BeagleBoard. *BeagleBoard.* [Online] BeagleBoard.org Foundation, August 2013. http://beagleboard.org/.

BricoGeek. 2013. BricoGeek. *BricoGeek.* [Online] BricoGeek, August 2013. http://www.bricogeek.com/shop/beaglebone/534-beaglebone.html.

Carriots. 2013. Carriots M2M Application Platform. *Carriots M2M Application Platform.* [Online] Carriots, S.L., August 2013. https://www.carriots.com/.

Carriots_b. 2013. Carriots M2M Application Platform. *Carriots M2M Application Platform.* [Online] Carriots, S.L., August 2013. https://www.carriots.com/documentation/carriots_ecosystem.

Carriots_c. 2013. Carriots | Project Management. *Carriots | Project Management.* [Online] Carriots, S.L., August 2013. https://www.carriots.com/documentation/api/project_management#projects.

Carriots_d. 2013. Carriots | Device Management. *Carriots | Device Management.* [Online] Carriots, S.L., August 2013. https://www.carriots.com/documentation/api/device_management#devices.

Catsoulis, John. 2005. *Designing Embedded Hardware.* s.l. : O'Reilly Media, Inc., 2005. 978-0-596-00755-3.

Coley, Gerald and Day, Robert P J. 2013. BeagleBone Black System Reference Manual. s.l. : beagleBoard.org, August 2013.

Coley, Gerald. 2012. BeagleBone Rev A6 System Reference Manual. s.l. : BeagleBoard.org, 2012.

Coley_b, Gerald. 2012. BeagleBoard-xM Rev C2 System Reference Manual. s.l. : BeagleBoard.org, 2012.

Coley_c, Gerald. 2012. BeagleBoard System Reference Manual Rev D. s.l. : BeagleBoard.org, 2012.

Davis, Jeff. 2010. *SOA Open Source.* s.l. : EDICIONES ANAYA MULTIMEDIA (GRUPO ANAYA, S.A.), 2010. 978-84-415-2679-2.

Debortoli, Stefan, et al. 2012. Internet of Things Architecture IoT-A . *Project Deliverable D2.3 – Orchestration of distributed IoT service interactions.* 2012.

ElectroSchematics. 2013. ElectroSchematics. *ElectroSchematics.* [Online] ElectroSchematics, August 2013. http://www.electroschematics.com/7960/arduino-leonardo-pinout/.

Erl, Thomas. 2008. *SOA Design Patterns.* Crawfordsville, Indiana (United States) : Prentice Hall, 2008. 978-0-13-613516-6.

EVRYTHNG. 2013. EVRYTHNG Make products smart. *EVRYTHNG Make products smart.* [Online] EVRTTHNG, August 2013. http://www.evrythng.com/.

EVRYTHNG_b. 2013. EVRYTHNG. *EVRYTHNG.* [Online] EVRYTHNG, August 2013. https://dev.evrythng.com/dashboard.

EVRYTHNG_c. 2013. API Reference. *API Reference.* [Online] EVRYTHNG, August 2013. https://dev.evrythng.com/documentation/api.

GigaMegaBlog. 2013. GigaMegaBlog - Beaglebone Coding 101: Blinking an LED. *GigaMegaBlog - Beaglebone Coding 101: Blinking an LED.* [Online] GigaMegaBlog, August 2013. http://www.gigamegablog.com/2012/01/05/beaglebone-coding-101-blinking-an-led/.

Gouvea, William. 2010. Let the Computer Decide. *Let the Computer Decide.* [Online] Blogspot, August 2010. http://amsterdaintelligence.blogspot.com.es/2010/08/oracle-pode-ter-uma-arma-secreta-na-mao.html.

Halfacree, Gareth and Upton, Eben. 2012. *Raspberry Pi User Guide.* s.l. : John Wiley & Sons, 2012. 978-1-118-46446-5.

ioBridge. 2013. ioBridge Connect Things. *ioBridge Connect Things.* [Online] ioBridge, Inc., August 2013. http://www.iobridge.com/.

ITU. 2005. The Internet of Things Executive Summary. 2005.

Kridner, Jason and Coley, Gerald. 2013. BeagleBone Black open-source Linux ™ computer unleashes innovation. s.l. : Texas Instruments, 2013.

Libelium. 2013. libelium. *libelium.* [Online] Libelium Comunicaciones Distribuidas S.L, August 2013. http://www.libelium.com/.

Lucas Martínez, Néstor. 2013. *Virtualización de dispositivos para la Ciudad del Futuro.* Madrid (Spain) : Final Project Dissertation, Advisor, José-Fernán Martínez Ortega: Universidad Politécnica de Madrid, 2013.

Martínez Ortega, José Fernán, et al. 2010. *Wireless sensor networks in knowledge management.* Madrid : Procedia Computer Science, 2010. Vol. 1. 1877-0509.

Microsoft. 2013. Microsoft Developer Network. *Microsoft Developer Network.* [Online] Microsoft, August 2013. http://msdn.microsoft.com/en-us/library/jj646625%28v=vs.102%29.aspx.

Mono. 2013. Mono. *Mono.* [Online] Xamarin, August 2013. http://www.mono-project.com/Main_Page.

netduino. 2013. netduino. *netduino.* [Online] netduino, August 2013. http://www.netduino.com/hardware/.

nimbits. 2013. nimbits.com. *nimbits.com.* [Online] Nimbits, Inc., August 2013. http://www.nimbits.com/.

Open.Sen.se. 2013. sen.se Feel. Act. Make sense. *sen.se Feel. Act. Make sense.* [Online] Sen.se, August 2013. http://open.sen.se/.

Oracle. 2013. Oracle. *Oracle.* [Online] Oracle, Inc., August 2013. http://www.oracle.com/index.html.

PCMAG. 2013. PC MAG. *PC MAG.* [Online] Ziff Davis, Inc, August 2013. http://www.pcmag.com/article2/0,2817,2407058,00.asp.

Pfister, Cuno. 2011. *Getting Started with the Internet of Things.* s.l. : O'Reilly Media, Inc., 2011. 978-1-4493-9357-1.

Postech. 2013. Mobile Networking Laboratory - Dept of Computer Science & Engineering, Division of IT Convergence Engineering. POSTECH. [Online] POSTECH, August 2013. http://monet.postech.ac.kr/research.html.

RaspberryPi. 2013. Rasberry Pi. *Rasberry Pi.* [Online] Rasberry Pi, August 2013. http://www.raspberrypi.org/.

RedSAM. 2013. REDSAM CONNECTING TECHNOLOGY. *RedSAM.* [Online] RedSAM, August 2013. http://www.redsam.nl/hardware/beagleboard-org-hardware-xm/.

Richardson, Matt and Wallace, Shawn. 2012. *Getting Started with Raspberry Pi.* s.l. : O'Reilly Media, Inc., 2012. 978-1-4493-4421-4.

RobotShop. 2013. RobotShop Putting robotics at your service. *RobotShop Putting robotics at your service.* [Online] RobotShop Distribution Inc., August 2013. http://www.robotshop.com/blog/en/arduino-5-minute-tutorials-lesson-2-basic-code-blink-led-2-3639.

Rodríguez-Molina, Jesús. 2012. *Semantic middleware development for the Internet of Things.* Madrid (Spain) : Final Project Dissertation, advisor, José Fernán Martínez Ortega; Universidad Politécnica de Madrid, 2012.

RPiHardware. 2013. RPi Hardware. *RPi Hardware.* [Online] RPi Hardware, August 2013. http://elinux.org/RPi_Hardware.

SmartSell. 2013. RESTful API in PHP for iPhone, Android and Web. *RESTful API in PHP for iPhone, Android and Web.* [Online] SmartSell, August 2013. http://www.smartsell.nl/restful-api-in-php-for-iphone-android-and-web/.

Sosinsky, Barrie. 2011. *Cloud Computing Bible.* s.l. : John Wiley & Sons, 2011. 978-0-470-90356-8.

Sun-Labs. 2010. Sun™ SPOT Programmer's Manual. s.l. : Sun Oracle, 2010. Vol. Release v6.0 (Yellow).

SunSPOTworld. 2013. Sun SPOT World. [Online] Oracle, Inc., August 2013. http://www.sunspotworld.com/.

Taylor, Hugh, et al. 2009. *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise.* s.l. : Addison-Wesley Professional, 2009. 978-0-321-32211-1.

ThingSpeak. 2013. ThingSpeak. *ThingSpeak.* [Online] ioBrige, Inc., August 2013. https://www.thingspeak.com/.

ThingSpeakb. 2013. ThingSpeak. *ThingSpeak.* [Online] ioBridge, Inc., August 2013. http://community.thingspeak.com/documentation/apps/thingtweet/.

ThingSpeakc. 2013. ThingSpeak. *ThingSpeak.* [Online] ioBridge, Inc., August 2013. http://community.thingspeak.com/documentation/apps/thinghttp/.

ThingSpeakd. 2013. ThingSpeak. *ThingSpeak.* [Online] ioBridge, Inc., August 2013. http://community.thingspeak.com/documentation/api/#thingspeak_api.

ThingSpeake. 2013. ThingSpeak. *ThingSpeak.* [Online] ioBridge, Inc., August 2013. https://www.thingspeak.com/channels/9.

trastejant. 2013. trastejant. *trastejant.* [Online] trastejant, August 2013. http://www.trastejant.es/proyectos/arduinodue.php.

Vermesan, Ovidiu, et al. 2010. Internet of Things Strategic Research Roadmap. 2010.

Walker, Chris. 2012. *Getting Started with Netduino.* s.l. : O'Reilly Media, Inc., 2012. 978-1-4493-0245-0.

Watts, Dan. 2013. GigaMegaBlog. *GigaMegaBlog.* [Online] WordPress, August 2013. http://www.gigamegablog.com/2012/01/05/beaglebone-coding-101-blinking-an-led/.

Wheat, Dale. 2011. *Arduino Internals.* s.l. : Apress, 2011. 978-1-4302-3882-9.

Xively. 2103. Xively The Internet of Things is open for business. *Xively The Internet of Things is open for business.* [Online] LogMeIn, Inc., August 2103. https://xively.com/.

Xively_b. 2013. Xively REST API. *Xively REST API.* [Online] LogMeIn, Inc., August 2013. https://xively.com/dev/docs/api/.

Xively_c. 2013. Create a Product - Xively. *Create a Product - Xively.* [Online] LogMeIn, Inc., August 2013. https://xively.com/dev/docs/api/product_management/products/create_product/.

Xively_d. 2013. Activate a device - Xively. *Activate a device - Xively.* [Online] LogMeIn, August 2013. https://xively.com/dev/docs/api/product_management/devices/activate_device/.

Xively_e. 2013. Single Datastream - Xively. *Single Datastream - Xively.* [Online] LogMeIn, Inc., August 2013. https://xively.com/dev/docs/api/data/read/single_datastream/.