E.U.I.T. TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA
PLAN 2000

TEMA:	Redes y Servicios Ubicuos para el Internet de las Cosas.	
TÍTULO:	Semantic middleware development for the Internet of Things.	
AUTOR:	Jesús Rodríguez Molina.	
TUTOR:	Dr. José Fernán Martínez Ortega.	**VºBº.**
DEPARTAMENTO:	DIATEL	
Miembros del Tribunal Calificador:		
PRESIDENTE:	Eduardo Juárez Martínez.	
VOCAL:	José Fernán Martínez Ortega.	
VOCAL SECRETARIO:	Vicente Hernández Díaz.	
Fecha lectura:		
Calificación:		**El Secretario,**

RESUMEN DEL PROYECTO:

Esta memoria comprende la realización del análisis, diseño, implementación y validación de una arquitectura middleware para el Internet de las Cosas, la cual se apoya en las aportaciones previas de otros investigadores de la escuela.

Partiendo de un estado del arte que sirva para adquirir los conocimientos necesarios para comprender los objetivos del resto del proyecto, se realizará un exhaustivo estudio de la arquitectura nSOM, una especificación middleware ubicua y orientada a servicios desarrollada por la EUITT, especialmente de los agentes software que han sido extendidos (Broker) y de los nuevos agentes software implementados (Orquestador, Alarmas del Orquestador), entre otros aspectos.

Este Proyecto Fin de Carrera ha sido desarrollado dentro del marco de trabajo del Proyecto de Investigación *LifeWear* (TSI-040200), cuyo consorcio ha obtenido el sello europeo ITEA2.

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

Semantic middleware development for the Internet of Things

Autor
Jesús Rodríguez Molina

Tutor
José Fernán Martínez Ortega
Dr. Ingeniero de Telecomunicación

Septiembre 2012

It matters not how strait the gate,

How charged with punishments the scroll.

I am the master of my fate:

I am the captain of my soul.

Extract from *Invictus*, by William Ernest Henley.

ACKNOWLEDGMENTS

I would like to acknowledge all the following people for their assistance and encouragement; they all have been of great help, either by providing their know-how, their direction or their moral support. Without them, having this Final Degree Dissertation made would have been much more difficult.

My parents for their tireless support, even at the moments when, academically, everything seemed to be lost, or I toyed with the idea of giving everything up. Fortunately or not, I believe I have already released them from that effort.

My friends inside and outside the EUITT, some of them a WhatsApp message away, others scattered throughout Europe. I firmly believe that all that lies ahead for them is a promising future achieving their objectives, regardless of how tough they are.

My Diatel colleagues Huang, Sandra and Pedro, who are always friendly, resourceful and skilful and have proved their value as workers and people in all the challenges we have faced. We have worked closely together in the research project of Lifewear where this Final Degree Dissertation is included, enriching us both technically and personally. Hopefully, in the near future we will carry on engaging in work duties together.

My professor and tutor Dr. José Fernán Martínez Ortega, for giving me the chance to work in the field of University Research and Development, which I am interested in above all the others, and the lecturers that have helped me throughout the Lifewear project: Vicente Hernández Díaz, Gregorio Rubio Cifuentes and professor Dr. Ana Belén García Hernando.

Last but not least, I would also like to acknowledge José Canfrán for letting me assist to his Final Degree Dissertation presentation and get a little grasp of what I could expect from it.

ABSTRACT

After the extraordinary spread of the World Wide Web during the last fifteen years, engineers and developers are pushing now the Internet to its next border. A new conception in computer science and networks communication has been burgeoning during roughly the last decade: a world where most of the computers of the future will be extremely downsized, to the point that they will look like dust at its most advanced prototypes. In this vision, every single element of our “real” world has an intelligent tag that carries all their relevant data, effectively mapping the “real” world into a “virtual” one, where all the electronically augmented objects are present, can interact among them and influence with their behaviour that of the other objects, or even the behaviour of a final human user. This is the vision of the Internet of the Future, which also draws ideas of several novel tendencies in computer science and networking, as pervasive computing and the Internet of Things.

As it has happened before, materializing a new paradigm that changes the way entities interrelate in this new environment has proved to be a goal full of challenges in the way. Right now the situation is exciting, with a plethora of new developments, proposals and models sprouting every time, often in an uncoordinated, decentralised manner away from any standardization, resembling somehow the *status quo* of the first developments of advanced computer networking, back in the 60s and the 70s. Usually, a system designed after the Internet of the Future will consist of one or several final user devices attached to these final users, a network –often a Wireless Sensor Network- charged with the task of collecting data for the final user devices, and sometimes a base station sending the data for its further processing to less hardware-constrained computers. When implementing a system designed with the Internet of the Future as a pattern, issues, and more specifically, limitations, that must be faced are numerous: lack of standards for platforms and protocols, processing bottlenecks, low battery lifetime, etc.

One of the main objectives of this project is presenting a functional model of how a system based on the paradigms linked to the Internet of the Future works, overcoming some of the difficulties that can be expected and showing a model for a middleware architecture specifically designed for a pervasive, ubiquitous system.

This Final Degree Dissertation is divided into several parts. Beginning with an Introduction to the main topics and concepts of this new model, a State of the Art is offered so as to provide a technological background. After that, an example of a semantic and service-oriented middleware is shown; later, a system built by means of this semantic and service-oriented middleware, and other components, is developed, justifying its placement in a particular scenario, describing it and analysing the data obtained from it. Finally, the conclusions inferred from this system and future works that would be good to be tackled are mentioned as well.

RESUMEN

Tras el extraordinario desarrollo de la Web durante los últimos quince años, ingenieros y desarrolladores empujan Internet hacia su siguiente frontera. Una nueva concepción en la computación y la comunicación a través de las redes ha estado floreciendo durante la última década; un mundo donde la mayoría de los ordenadores del futuro serán extremadamente reducidas de tamaño, hasta el punto que parecerán polvo en sus más avanzado prototipos. En esta visión, cada uno de los elementos de nuestro mundo “real” tiene una etiqueta inteligente que porta sus datos relevantes, mapeando de manera efectiva el mundo “real” en uno “virtual”, donde todos los objetos electrónicamente aumentados están presentes, pueden interactuar entre ellos e influenciar con su comportamiento el de los otros, o incluso el comportamiento del usuario final humano. Ésta es la visión del Internet del Futuro, que también toma ideas de varias tendencias nuevas en las ciencias de la computación y las redes de ordenadores, como la computación omnipresente y el Internet de las Cosas.

Como ha sucedido antes, materializar un nuevo paradigma que cambia la manera en que las entidades se interrelacionan en este nuevo entorno ha demostrado ser una meta llena de retos en el camino. Ahora mismo la situación es emocionante, con una plétora de nuevos desarrollos, propuestas y modelos brotando todo el rato, a menudo de una manera descoordinada y descentralizada lejos de cualquier estandarización, recordando de alguna manera el estado de cosas de los primeros desarrollos de redes de ordenadores avanzadas, allá por los años 60 y 70. Normalmente, un sistema diseñado con el Internet del futuro como modelo consistirá en uno o varios dispositivos para usuario final sujetos a estos usuarios finales, una red –a menudo, una red de sensores inalámbricos- encargada de recolectar datos para los dispositivos de usuario final, y a veces una estación base enviando los datos para su consiguiente procesado en ordenadores menos limitados en hardware. Al implementar un sistema diseñado con el Internet del futuro como patrón, los problemas, y más específicamente, las limitaciones que deben enfrentarse son numerosas: falta de estándares para plataformas y protocolos, cuellos de botella en el procesado, bajo tiempo de vida de las baterías, etc.

Uno de los principales objetivos de este Proyecto Fin de Carrera es presentar un modelo funcional de cómo trabaja un sistema basado en los paradigmas relacionados al Internet del futuro, superando algunas de las dificultades que pueden esperarse y mostrando un modelo de una arquitectura middleware específicamente diseñado para un sistema omnipresente y ubicuo.

Este Proyecto Fin de Carrera está dividido en varias partes. Empezando por una introducción a los principales temas y conceptos de este modelo, un estado del arte es ofrecido para proveer un trasfondo tecnológico. Después de eso, se muestra un ejemplo de middleware semántico orientado a servicios; después, se desarrolla un sistema construido por medio de este middleware semántico orientado a servicios, justificando su localización en un escenario particular, describiéndolo y analizando los datos obtenidos de él. Finalmente, las conclusiones extraídas de este sistema y las futuras tareas que sería bueno tratar también son mencionadas.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF CHARTS

ACRONYMS

Acronym	Extended text
6loWPAN	IPv6 over Low power Wireless Personal Area Networks
ACSE	Association Service Control Element
ADC	Analog to Digital Converter
AEMET	Agencia Estatal de METeorología
AES	Advanced Encryption Standard
AFH	Adaptive Frequency-Hopping spread spectrum
API	Application Programming Interface
ARP	Address Resolution Protocol
AURA	application-aware and user-interaction aware middleware platform
BAN	Body Area Network
BPEL	Business Process Execution Language
BSD	Berkeley Software Distribution
CASAGRAS	Coordination and support action for global RFID-related activities and standardization
CLDC	Connected Limited Device Configuration
CMIDSE	Common Medical Device Information Service Element
CoAP	Constrained Application Protocol
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CORDIS	Community Research and Development Information Service
CPU	Central Processing Unit
DAML+OIL	Defense Advanced Research Projects Agency Agent Markup Language + Ontology Inference Layer
DCDO	Disconnection Code Description Object
DCOM	Distributed Component Object Model
DIO	Digital Input/ Output
DIO	Device Identifier Object
DMNAO	Device Model Name Object
DMNUO	Device Model Number Object
DMO	Device Manufacturer Object
DMURLO	Device Model URL Object
DNO	Device Name Object
DSNO	Device Serial Number Object
DTDO	Device Type Description Object
EDO	Event Description Object
EEPROM	Electrically Erasable Programmable Read-Only Memory
ESB	Enterprise Service Bus
EUITT	Escuela Universitaria de Ingeniería Técnica de telecomunicación
FSK	Frequency-Shift Keying
GNU	GNU´s not UNIX!

GPIO	General Purpose Input/ Output
GRyS	Grupo de Redes y Servicios de Próxima Generación
HTTP	Hyper Text Transfer Protocol
I²C	Inter-Integrated Circuit
I²S	Integrated Interchip Sound
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IEEE	Institute of electrical and Electronic Engineers
IoT	Internet of Things
IP	Internet Protocol
ISM	Industrial, Scientific and Medical
ISO	International Organization for Standardization
J2ME	Java 2 Micro Edition
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
MAC	Medium Access Control
MANET	Mobile and Ad-hoc connected NETwork
MARKS	Middleware Adaptability for Resource Discovery, Knowledge Usability and Self-healing
MDIB	Medical Data Information Base
MDP	Markov Decision Process
MiLAN	Middleware Linking Applications and Networks
MOM	Message-Oriented Middleware
nesC	Network Embedded Systems C
nSOM	nano Service Oriented Architecture
OSI	Open Systems Interconnection
OWL	Web Ontology Language
PAN	Personal Area Network
PC	Personal Computer
PCI	Protocol Control Information
PCOM	Component System for Pervasive Computing
PDA	Personal Digital Assistant
PDU	Protocol Data Unit
PHD	Personal Health Device
PWM	Pulse Width Modulation
QoS	Quality of Service
QPSK	Quadrature Phase-Shift Keying
RAM	Random Access memory
RCSM	Reconfigurable Context Sensitive Middleware
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
REST	REpresentational State Transfer
RFID	Radio Frequency IDentification

RIF	Rule Interchange Format
RMI	Remote Method Indication
ROM	Read Only Memory
ROSE	Remote Operation Service Element
RPC	Remote Procedure Call
RPL	Routing Protocol for Low power and Lossy Networks
RTOS	Real Time Operating System
SD	Secure Digital
SDO	Service Description Object
SDU	Service Data Unit
SEO	Service Event Object
SFS	Self-certifying File System
SICS	Swedish Institute of Computer Science
SIO	Service Identifier Object
SME	Small and Medium-sized Enterprise
SOA	Service Oriented Architecture
SOAP	Simple Object Access protocol
SOC	Service Oriented Computing
SOIO	Service Operation Invocation Object
SPARQL	SPARQL Protocol and RDF Query Language
SPI	Serial Peripheral Interface
SQL	Structured Query Language
SQO	Service Query Object
SSDO	Service Subscription Description Object
SSDP	Simple Service Discovery Protocol
SunSPOT	Sun Small Programmable Object Technology
SWRL	Semantic Web Rule Language
TCP	Transport Control Protocol
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
UPM	Universidad Politécnica de Madrid
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
USB	Universal Serial Bus
VNC	Virtual Network Computing
W3C	World Wide Web Consortium
Wi-Fi	Wireless Fidelity
WSAN	Wireless Sensor and Actuator Network
WSDL	Web Service Description Language
WSN	Wireless Sensor Network
xDSL	x [Asymmetric, Symmetric, etc.] Digital Subscriber Line

XML	eXtensible Markup Language

Chapter 1

Introduction and objectives

1.1 The concept of ubiquitous computing.

It was Mark Weiser, now considered the forerunner and father of this field of computer science who first coined the term “ubiquitous computing” [Berkeley 99]. In his view, tiny computers would be added to all the common objects, utilities and tools used in a daily routine (from blankets to televisions, machinery, furniture, etc) to the point that this tiny computer grid would become “ubiquitous” and “pervasive”, that is, be present simultaneously everywhere within an area that can be as big as the very planet Earth, or even beyond. This tiny computers, better considered as nodes of a network, harvest information from the entities they are attached to, distributing and sharing it either with other components of the grid or sending them to a final human user, depending on the nature of the service that is going to be provided. In a way, ubiquitous computing can be compared to the top of an iceberg: it is possible because there are many other developments and technologies below (in a layered architecture fashion) and behind (in a timeline) it. In Mark Weiser´s own words: “*Ubiquitous computing names the third wave in computing, just now beginning. First were mainframes, each shared by lots of people. Now we are in the personal computing era, person and machine staring uneasily at each other across the desktop. Next comes ubiquitous computing, or the age of calm technology, when technology recedes into the background of our lives*” [Berkeley 99].

Ubiquitous computing has also been characterized by J. York and P.C. Pendharkar as “*machines that fit the human environment instead of forcing humans to enter theirs*” [York 03]. This definition has an important shade of meaning: ubiquitous tiny computers scattered in an environment will demand as little user interaction as possible; rather than providing human-to-machine interfaces for communication with human users, they will rely just on machine-to-machine interfaces to interact with other ubiquitous computers without annoying people. In any case, given that they are very small, it would be impractical to provide then with keypads or monitors, and the final user would not be able to pay attention to all the data collected at the same time.

Commonly, a ubiquitous architecture will be consisting of several pieces of hardware, that is, the ubiquitous computers that will be operating as nodes. These nodes usually belong to either a Wireless Sensor Network or WSN (occasionally, they can be named as WSAN, Wireless Sensor and Actuator Networks, to stress the idea that there may be actuators taking actions based on the content instead of just harvesting it) or they are pieces of equipment attached to the final user (in fact, if they have several pieces of equipment on them, they might be composing a Personal/Body Area Network or PAN/BAN). Over this wireless network level there will be a middleware architecture receiving all the gathered content and abstracting all the heterogeneity of the Wireless Sensor Network components, turning content into homogeneous information units that can be easily managed by an application with human-to-machine access interfaces for the final user. In the end, components will be deployed as the way they are shown on next page:

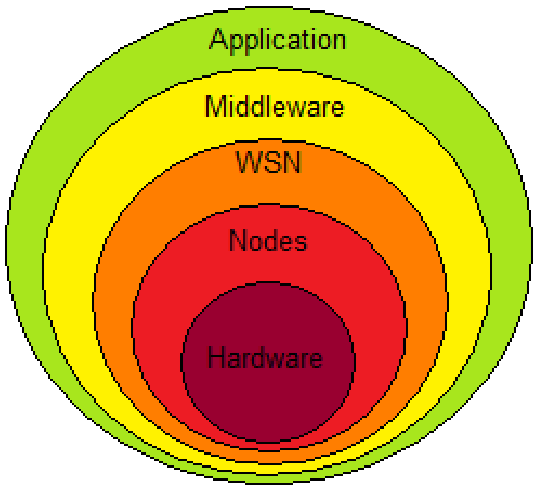


Figure 1: a model showing the places where every component is placed.

1.1.1 Foundations of ubiquitous computing.

Although ubiquitous computing can be described or found under different names (pervasive computing, everyware) its main features were suggested by Mark Weiser, and thus virtually all the other given names refer to the same core concepts:

1. **"*The purpose of a computer is to help you do something else*"**. As it has been asserted before, computers in a ubiquitous environment should remain unnoticed. The concept of usage of a computer under these new rules differs from the traditional wired or wireless networks: the computer is not accessed explicitly by a human so as to obtain a service, but ubiquitous computers establish connections and communications among themselves in order to silently guarantee the provisioning of a service. Thus, the computer is not self-centred, and actually, requires the interaction with other computers –or more likely, nodes- from the wireless network to achieve its part of the demanded service.

2. **"*The best computer is a quiet, invisible servant*"**. If, for example, a user does not demand constant interactions with a lamp or a telephone, the lamp or the telephone will remain idle and will not require any attention from the user. So, in the case they were electronically augmented, their operational behaviour should remain the same: computers attached to everyday´s elements must remain unnoticed and latent, even when they are providing any service (as ubiquitous computers cooperate to offer a service instead of having the final user to ask for it on a particular terminal), with the final aim of offering a holistic integration with the environment, where electronically-augmented devices are just like the non-augmented regular ones, without any other further notification for the final user.

3. ***"The more you can do by intuition the smarter you are; the computer should extend your unconscious"***. True holistic, seamless integration of ubiquitous components, as if they were

interweaved with the environment, requires that usage and interaction with them is done in a natural and unconscious-like manner, the same way driving or other activities as reading and writing are done.

4. ***"Technology should create calm"***. Meaning by this statement that technology should provide a flow of relevant information to the final user but not in an intrusive and noisy fashion (asking for user intervention every other time, transcending the usual functionalities of the object the ubiquitous computer is embedded in, etc.)

In addition to those principles formulated by Mark Weiser, several other ideas can be added according to the experience gained by all the developments made up to this date:

5. An **ambient intelligence** can be enabled. Thanks to the data retrieval and due to the fact that same requests are managed the same repeated way, the environmental nodes of an architecture can come to gain some intelligence, meaning by that learning how to best tackle a task; for example, if sensors are present in a smart home and the system has a profile associated for each of the residents, when one of them arrives the system can foresee what time the resident will be taking a bath or what luminosity level prefers after having the resident stating their requests with the same parameters several times.

6. **Context awareness** is present. Somewhat linked to the idea of an ambient intelligence, context awareness is what allows a ubiquitous system to acknowledge under which precise and current conditions the system is carrying out its main duties. This is usually achieved by the readings given by the sensors of system; for example, if there is a system in charge of watering a variety of crops, if it is able to distinguish between a sunny or a rainy day (and therefore, the varying amount of water required by the plants) or what crop is the one they are monitoring (wheat, for example, will not require the amount of water that corn does), then the system is context-aware.

7. **User intention** can be easily inferred. As it happened on the two features before this, if a task is repeatedly done the same way through many times under similar conditions, the system will be able to deduct what next task the final user is going to try; going on with the smart home example, if a resident arrives at a certain hour and likes watching a TV series on Thursday night, the system can anticipate to the resident and suggest that service by its own; the system could even react to TV timetable changes in order to keep the resident informed about any variation on the expected parameters.

8. **Wireless Sensor Networks are a cornerstone** for developing ubiquitous systems. Because of their abilities to locate sensors in almost every imaginable place and the low power-consuming devices they use to provide services, Wireless Sensor Networks are usually heavily implicated in ubiquitous systems. More often than not, Wireless Sensor Networks will have some sort of gateway between the ubiquitous environment and a wired, more conventional one that will be

used to share the information either with the final user or with other systems with a different paradigm.

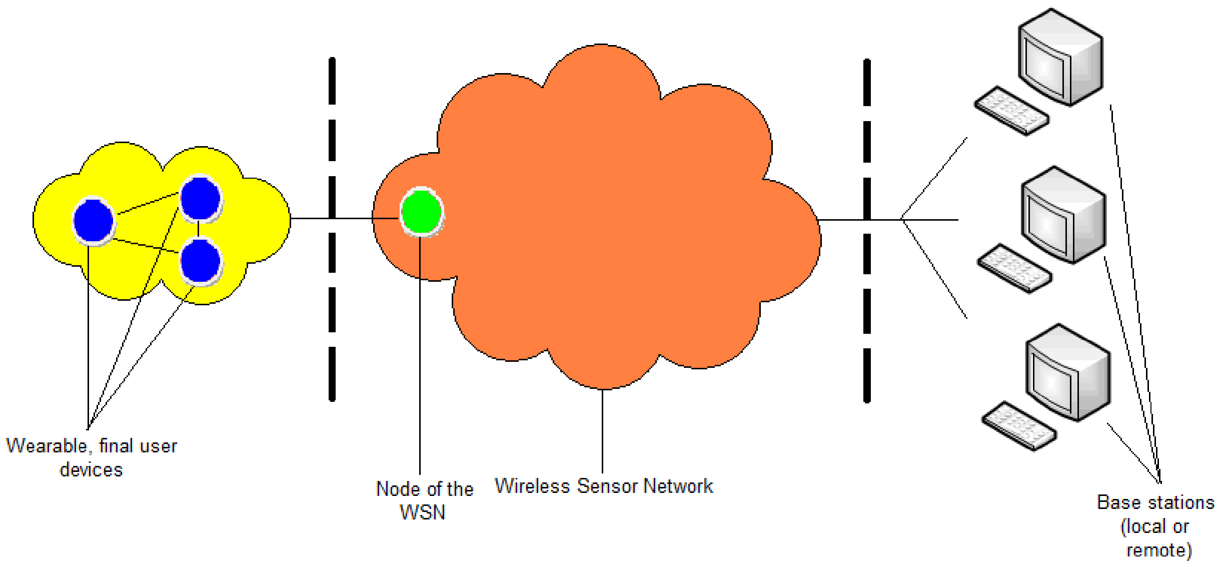


Figure 2: a Wireless Sensor Network architecture.

1.1.2 Challenges of ubiquitous computing.

As this is a new field in communications, there are new issues and constrains that were not present before. Usually, they appear around the same key points:

- Nodes of the Wireless Sensor Networks implied in the communications are not like regular routers of Wireless Access Points: they have severe limitations related to their inherent nature of wireless nodes (power is not unlimited in time and energy cannot be easily harvested, radio transmissions have to be reduced and their area of coverage shrunk). Also, the most widespread nodes –or more accurately, motes- that are marketed, being way smaller than their wired counterparts, are not as small as to be ignored once they are attached to an object, so the seamless integration of them is a task yet to be achieved with affordable costs.

- Related with costs comes up the problem of having a Wireless Sensor Network ranging from the dozens to the thousands of nodes, so any increase in the cost of the motes, as small as it may be, may have a significant impact in the cost of the whole network if it is replicated for all the other motes. And should the motes be from different manufacturers, there may be additional trouble implementing a satisfactory middleware architecture that can accomplish its expected functionalities.

- Commonly, middleware architectures are not built to implement a brand new model in ubiquitous computing but they are usually designed after a very specific need, thus preventing any real standardization of all the proposals of middleware architectures. This is a serious issue because every time that a middleware architecture has to be put forward in almost has to be built from scratch, or at least using developments of earlier architectures that are not necessarily done under the premises of ubiquitous computing. This is just a symptom of the immaturity of many middleware proposals, which in fact until very little time ago were not built after specifically thought ubiquitous environments.

- Compared to a "conventional" scenario, a ubiquitous architecture is usually less reliable and more prone to errors and node failures. This is due to the ever-shifting nature not only of the services (they can be dependent on nodes that may be out of commission, or the data required by them to work can be outdated) but also to the fact that the final devices that are carried by the user have much more mobility than the conventional ones, who are fixed on a very precise spot of a room. Self-healing mechanisms are even more welcomed here than in conventional networking.

- Traditional communication models are difficult to be used here; other alternative models based on agents, which will provide some autonomy over the middleware and the infrastructure below, or based on events, that will trigger some nodes or some services, may come in handy for future developments.

1.2 Motivations, objectives and contributions.

The motivations behind a Final Degree Dissertation like this grow from the idea that a contribution to the developing field of the Internet of the Future, in a moment when technologies and developments in this field are not rock solid yet, and therefore can be shaped to an extent with the work or researchers, developers and engineers, is something stimulating and rewarding. An acquisition of know-how and experience in the fields related to the system that will be presented later (software engineering, programming, embedded systems) are desirable too.

Besides, the leading objectives of this Final Degree Dissertation are:

- Undertaking a profound State of the Art regarding middleware architectures, not only to get a grasp of the middleware architectures that have been used in the past, but also of the different alternatives of the present, along with the open issues that challenge the development of legacy and avant-garde middleware subsystems. Besides, a survey on novel tendencies in middleware computing is shown for its usability value.

- Making use of EUITT´s semantic and wearable-oriented middleware architecture (nSOM) to, after describing it thoroughly and making clear all the user component´s and behaviour, create a system where the principles of pervasive, ubiquitous and wearable computing are put into practice, especially regarding services composition, network deployment and the information that can be retrieved from them.

- Using the possibilities that a lightweight service description language as JSON (JavaScript Object Notation) offers to provide and study the communication performed between nodes and elements from a Wireless Sensor Network, along with its most important parameters (network deployment, simple and composed services, differences between unicast, multicast and broadcast messages, etc.).

- Hinting what additions and improvements could be codified into the nSOM semantic middleware architecture for future developments, given the experience acquired after its usage in a real scenario.

1.3 Technological framework.

This Final Degree Dissertation is encased within the research project “LifeWear - Mobilized Lifestyle with Wearables” [Lifewear 12b] (ITEA2, TSI-040200). This project is funded by the program Avanza Competitividad from The Spanish Ministry of Industry, Tourism and Commerce, with partners from both inside Spain and Europe. The objectives of this project are increasing the quality of life of a user by the usage of wearable sensors and devices [Lifewear 10], as well as making wearable devices, and their supporting technologies, more acceptable and popular in everyday´s life.

1.4 Structure of the report.

This Final Degree Dissertation can be broken into several sections in order to make the extensive considerations and concepts on it more manageable:

- **Chapter 1**: here, an introduction has been made on the concepts that are pivotal to the understanding of this Final Degree Dissertation: wearable computing, ubiquitous and pervasive computing, where their foundations lie or the special conditions and challenges to be dealt with when creating a system based on the proposals of wearable computing.

- **Chapter 2**: a state of the art in wearable-oriented middleware is shown here. A statement of the problem of porting middleware to these architectures in made first, followed of an engineering perspective, how the middleware is –and has been- managed and developed under these new circumstances and both what open issues are currently crippling the development of ubiquitous, pervasive middleware and how the new approaches regarding middleware are bent on tackling them.

- **Chapter 3**: a thorough exposition of the semantic middleware developed by the GRyS research group of the Universidad Politécnica de Madrid, called nSOM (nanoService Oriented Middleware). The architecture is exposed in a general overview, along with all its components one by one. Along with it, the distinctive functionalities that are offered, and the procedures used to offer them to the final user and the networking nodes when it is required are also described.

- **Chapter 4**: a system was deployed using a group of hardware (motes, PCs, etc.) and software components (an ESB, nSOM, etc.), citing all the system actors and the user cases this scenario would offer. In addition to this, a description of the validation scenario and an analysis of the validation scenario results were also provided.

- **Chapter 5**: a final chapter has been written noting all the conclusions that can be extracted from this development, along with the future works that would be interesting to carry out in the short and mid-term future.

- Finally, two **annexes** are provided, one with the most notorious APIs of the nSOM components and another one with a compelling chart of all the currently available nodes for a Wireless Sensor Network. Right after that, Bibliographic References are shown.

Chapter 2

State of the art in ubiquitous middleware

2.1 Problem statement.

It can be argued that adding a new layer on a model that already has a few of them will bring unwanted complexity to the whole system and new issues when trying to solve the relations between the layers that are kept either below or above the new one. Besides, middleware does not provide any tangible functionality by itself: it does not give shape to user interfaces, nor it does collect information from elements of the environments that are going to be studied. At a first glance, it just looks like a waste of time and resources.

However, this notion makes no sense for various reasons. First of all, nowadays computers –and especially, wearable, ubiquitous-related computers- are extremely unlikely to work isolated one from another. On the other hand, computers involved in a communication are extremely unlikely to be perfectly equal machines as well, so information obtained by them by any means will be shared according to how their hardware has been designed (for example, little endiand and big endiand figure representations) and finally, with no further intervention, it is extremely unlikely that results will be understandable for users and, before them, for computers involved in the communication. An additional layer used to adapt all the information coming from hardware-based layers and having it usable for any other computer of the network –a functionality assumed by middleware today- is a challenge that computer scientists and engineers had to face way before the dawn of the era of ubiquitous, pervasive middleware or architectures. In fact, the concept of middleware first appeared as early as 1968 [Ironick 05], referred to as "*software used to adapt generic file system functionality to specific application functionality needs*" and became popular later in the 1980s as a way to solve the issues inherent to link newer applications to old legacy systems [Weiderman&97].

Unfortunately, under the conditions where the systems that interest this Final Degree Dissertation perform their duties, a "regular" middleware will find it difficult to fit. Although reasons for claiming so will be developed later, they can be firstly introduced here: many non-ubiquitous middleware solutions belong to an environment created before the specific needs of ubiquitous and pervasive computing were formulated, or even conceived. Therefore, although some of their ideas can be borrowed, usually conventional middleware architectures are unable to be adapted into the environment mentioned before, leaving them powerless to accomplish their expected middleware functionalities.

Because of that, if applications and utilities are to be developed in the context of the Internet of the Future, Internet of Things or by using Wireless Sensor Networks, a ubiquitous, pervasive middleware must be implemented to ensure a correct performance of the whole system.

2.2 Engineering in middleware for wearable services.

Consequently, as a result of the issues mentioned before, another layer of software was created. How this layer works, its motivations and how it is applied to a ubiquitous environment is a great deal of what this chapter is going to be about.

2.2.1 Main objectives of middleware

Traditionally, middleware has retained a critical functionality within the domain of electronic systems and appliances -that is, separating the layers above middleware- with its presence justified by a) The need of further development of the software embedded in the actual device into applications and b) Human operators, from programmers to end users, need an high level layer to interact with the whole of the device- from the heterogeneity and peculiarities of the lower, more hardware-oriented layers, and by doing so, providing higher levels with an homogenous and abstract environment. This functionality is overall kept as far as wearable computing is concerned, taking into account the expectable adaptations that must be faced in order to have a middleware layer performing seamlessly under new conditions and rules

Besides, some other objectives can be considered as typical in middleware computing: since it enables developers to make use of a high abstraction level, they can focus on interactions between applications, as their communications are guaranteed by the middleware layer just by making sure that regardless of the system or the electronics put to a use physically, all this heterogeneity will be insulated, enabling interactions among applications on high levels easily (in a networking environment and according to [Marsden, 1991], middleware implements the Session and Presentation Layers of the OSI Reference Model.) as it is said in [Emmerich&02].

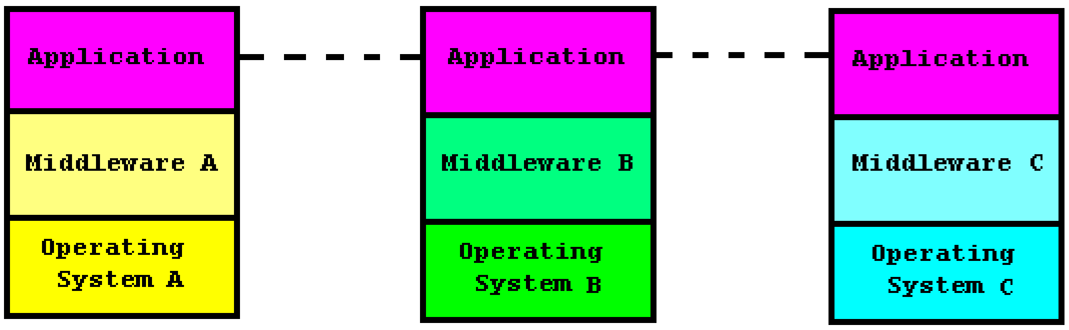


Figure 3: different kinds of middleware adapt the layers below them.

Given the particularities of wearable devices and ubiquitous computing, some adaptations will have to take place inevitably. These adaptations have been carried out given the foreseeable challenges that may be faced in a ubiquitous computing-oriented context:

- **Dramatic miniaturization of hardware and software characteristics.** Some already established devices which, in one way or another, are constrained to low power consuming or narrow bandwidth necessities, as motes in ubiquitous networking, are sure to be smaller than domestic computers, so services need to be provided at their minimum possible capabilities of transmission, energy storage, etc. This radical shrinking in hardware (and consequently, in software) will take its toll when creating new services and experiences, but without it wearable devices and ubiquitous computing would be meaningless.

- **Human intervention is preferably nonexistent**. As new paradigms in computing (Internet of Things, Service Oriented Computing) are developed under the ideas of Mark D. Weiser, it becomes crystal clear that if an everyday augmented element is to become part of a service provided by a Wireless Sensor Network, that supposedly is as quiet and discrete as it can be, intrusion and attention from a human being must be kept at their lowest rates (unless the end user is willing to be told about any datum related to the service) in order to have the wearable device turning into –or remain as- a silent, non-attention demanding electronic piece naturally integrated within its environment –or embedded in an object that is already like that-, which will not add extra duties or discomfort to its human beneficiaries.

- **Ever-shifting nature of the ubiquitous content.** Should OSI or TCP/IP architectures be considered as the components architecture of a service found in wearable computing, with a physical layer at the bottom and an application layer at the top, any wearable device is likely to find a huge number of its kind scattered through those layers: personal computers´, wireless nodes´ or smart phones´ electronics as hardware and almost any imaginable idea as applications. Not only must an efficient wearable-based computing middleware layer abstract the general working conditions and offer a usable interface to higher levels, but also has to deal with the changing status of applications: if the wearable device is working in combination with any piece of clothes, services will change as the user walks, drives, etc. In this way, some services will be lost, rendering the application built to exploit them useless, and some other services will be discovered, activating applications already existing or downloading new ones. Middleware should be designed properly to face this possibility and not leaving the user with only a set of applications available in just a few places.

Consequently, if middleware is going to run on machines designed for this ubiquitous context, it must fulfil a series of new requirements not found in a better-known area:

1. **High flexibility and adaptation skills.** Unlike conventional networks, whose number of members, configuration, topology and general features change from time to time in a slow pace, wearable devices and ubiquitous networks are way more dynamic, as it has been learnt. A middleware capable of disabling obsolete applications, dropping connections with unreachable nodes or, on the contrary, enabling new applications and discovering new pieces of hardware in the ubiquitous network is a must.

2. **Reusability for existing pervasive applications.** In an area changing its appearance and its utilities in an accelerated fashion, many applications (and the means to support them) could rapidly become outdated and, what is worse, unusable. One of the critical functionalities of the middleware layer will be making sure that a generous grade of compatibility and retro-compatibility is present and not just the latest versions of an application are the only functional ones (It has to be considered that many -if not most of- users will not renew them as soon as

an update or an upgrade is available). After all, one of the main reasons for designing early middleware platforms was the correct performance of new applications coexisting with old legacy systems [Saif&01].

3. **Interoperability among different platforms.** Obviously, if one of the most important missions of middleware is negating the complexity of the physical layer and providing a generic abstraction to engineers, the differences and boundaries between platforms that may be present will have to be dissolved so as to develop new applications which are not limited in their scope by the final hardware device that is behind them. Note that since hardware is primarily controlled by the operating system of each electronic device, any interaction with the former will imply requests to the latter, so the complexity of the different operating systems must be born in mind.

Basically, there are two different directions to get a grasp of how to develop middleware in this field: a regular vision (albeit increasingly deemed outdated) with legacy solutions in the ubiquitous computing area and another one based on emerging research results, such as Service Oriented Computing and Semantic Management Knowledge. In addition, a quick look at regular distributed middleware solutions will be taken, although efficiency and usefulness are thwarted by the failure in adapting to a ubiquitous environment.

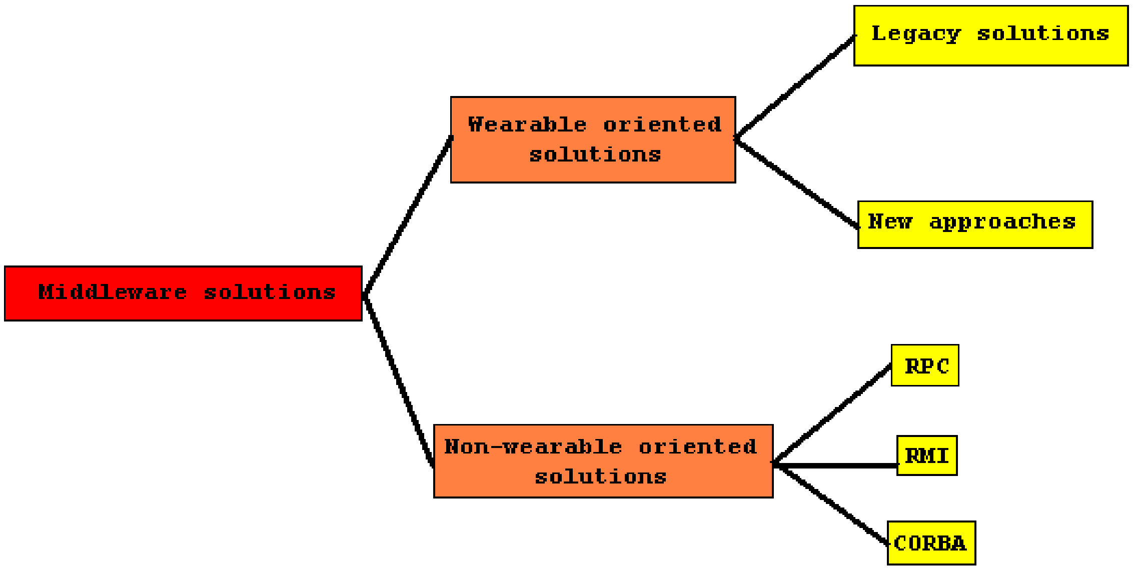


Figure 4: a taxonomy of the middleware solutions analysed in this state of the art

2.2.2 Requirements in wearable services

Several concepts have been suggested to achieve a satisfactory middleware solution even before the latest paradigms, like Service Oriented Computing and Semantic Knowledge, made their first appearance. The easiest was developing a port of already existing solutions to a ubiquitous environment, while other research projects suggested an entirely new system to cope with an entirely new area. It is interesting noting that as a middleware layer will be present in all the devices of a ubiquitous environment, middleware developments must be aware that whatever the device is (a wearable computer, a sensor node) other devices present in the network have to be considered.

Despite its peculiarities and different adaptations for each service, wearable-oriented middleware will be generally composed by the same several modules in almost all of the implementations:

- **A lower-layer communication module.** This is a foreseeable part of any conceivable middleware, as their leading mission is retrieving the heterogeneous data poured from lower layers with fragmented, different technologies (hardware, operating systems, etc) and turn it into a common format, comprehensive for the higher layers of any implemented device (in this case, a wearable appliance), as a regular Object Request Broker would do. Although either high or low layer may differ from what a traditional middleware would deal with, the underlying functionality remains the same.

- **A context manager module.** As it has already been explained, context is a core concept in ubiquitous computing, since it will determine not only whether a collection of services will be available or not, but also the precise moment of this availability –or unavailability-. In order to have an accurate idea of this fact, a context manager module is needed so that a categorization of the different contexts and an arranged list of services for each context will be composed. In the end, the context manager will deploy the reachable services for the final user as long as the current context allows them to be available.

- **A service manager.** Working closely with the context manager, once it has approved the suitable collection of services, the service manager will tackle the common actions related to the performance of a service: registry, login, logout, download or deletion. Plus, the service manager will be responsible for the discovery of existing services in a context, previously made known by the context manager.

- **A distributed data transfer module.** Once the input data (obtained from lower levels and converted into an understandable format by higher ones) has been processed, it will be transported to the ending point in charge of broadcasting the input data to the interested receiver. To accomplish this task, a module dedicated to the information transfer will be designed; nevertheless, this module will be related to a classic functionality of any sort of middleware, so it will not require a complete new development (except adaptations to a ubiquitous environment). Commonly, it will also provide information from the application layer that may come in handy for Context and Service managers.

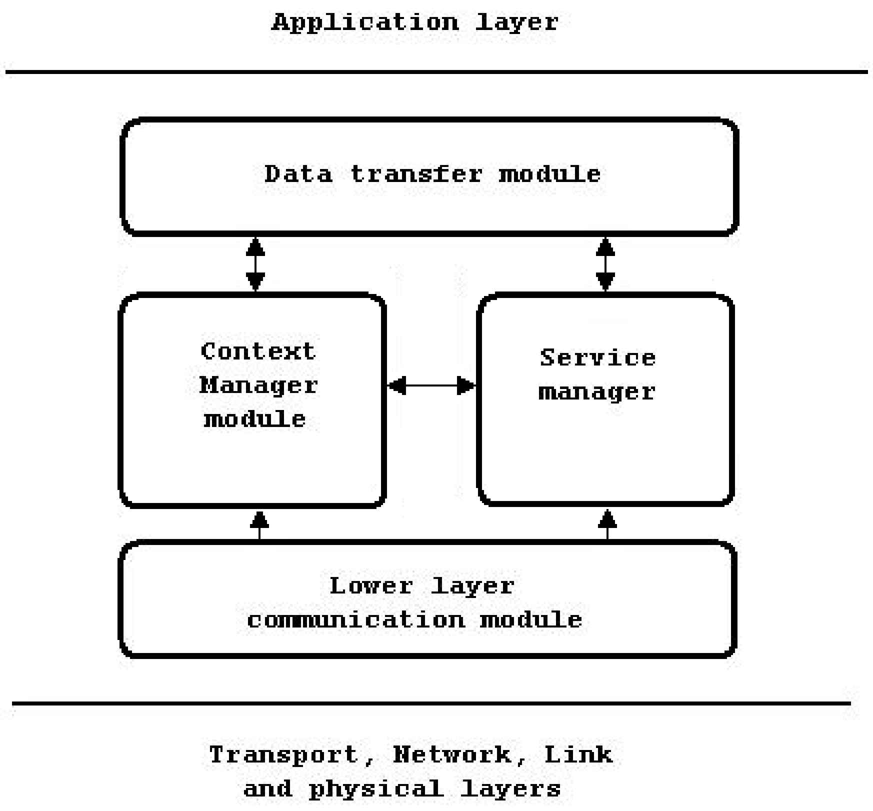

Application layer
Data transfer module
Context Manager module
Service manager
Lower layer communication module
Transport, Network, Link and physical layers

Figure 5: different modules of the middleware layers, along with their relationships among them.

In order to have a global yet detailed view of the most prominent aspects of the middleware architectures that are going to be reviewed, a classification of the main middleware ubiquitous platforms that have achieved some sort of relevance is offered below these lines, inspired by the one suggested as an orthogonal classification of middleware technologies by Noha Ibrahim [Ibrahim 09]. In this conception, there are two views comprised on the same classification:

- A "horizontal" one, where already developed and deployed middleware architectures are present in a layer called Middleware of Types, and new layers have been incorporated under –Middleware of Sensors- and over –"Middleware of Middlewares"- it, thus building a stack of differentiated middleware components.

- A "vertical" one, where a collection of features is gathered under self- or ubiquitous middleware; the focus will be driven to ubiquitous middleware, since self-middleware functionalities according to Ibrahim´s classification are either successfully dealt with in a ubiquitous environment by ubiquitous middleware, or out of the scope of this Final Degree Dissertation.

The resulting classification is as follows:

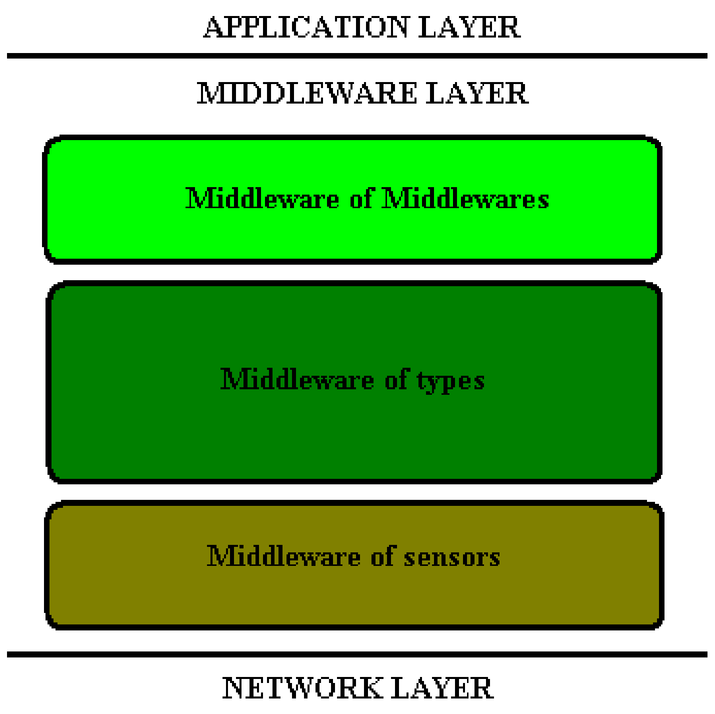


Figure 6: classification of the components for the middleware layer.

Middleware of Middlewares will offer the needed abstraction from the lower levels (in this case, the type of middleware itself), in a four-staged way, reminiscent of the needs of Service-Oriented Architecture or SOA: a first phase devoted to the development of the application, another one of deployment of the already developed application on the environment, a third phase for execution (which will be focused on the correct performance of the application) and a fourth one where the interaction capabilities of the application are taken care of.

The Middleware of Sensors deals with different suggestions about middleware commonly designed for Wireless Sensor Networks or WSNs. A middleware on this level will usually manage the issues related with the wireless network directly (for example, should a node have a failure, this layer will try to find another one to keep the data flowing) with minimal user interaction.

The Middleware of Types layer is where the different approaches that have become fruitful and turned into different middleware platforms, suited to ubiquitous environments, are located. This project will bend its efforts on a proposal encased in the Middleware of Types layer, so the following reviewed middleware architectures can be placed within the domain of this level.

2.2.3 Legacy solutions in middleware.

These solutions have been proven to be of great help in traditional networking (equally in wired and wireless elements, since they will behave the same way in a link, network, transport or application context) but they are effectively unusable in services with wearable devices and ubiquitous networks at both of its ends, mostly because these solutions were implemented well before the Internet of Things and Ubiquitous Computing paradigms were even conceived. Therefore, despite of faring quite well in environments where power supply is taken for granted or at least is reliable enough, and the nodes taking part of them being robust and well-known, when dealing with the new model, some issues almost impossible to get around spring up.

To begin with, Remote Procedure Call (RPC) has been discarded for a long time as an employable middleware in the context this Final Degree Dissertation is located. Among its multiple flaws, it is said about it that "*RPC semantics of a synchronous, blocking invocation on a statically typed interface are overly restrictive, inflexible, and fail to provide an efficient unifying abstraction for accessing and modifying state in ubiquitous systems*" [Saif&01]. The general feeling about RPC in the research community is that it is too rigid and unable to deliver a good performance in a dynamic behaviour as a ubiquitous environment demands; for example, it is unable to offer the result of a request to other host or process different from the one that originated it, something that could be needed if a service is to carry some data to a wearable device belonging to a Personal Area Network which is not the source of the request, or if the node that originated the request is down and the response has to be redirected somewhere else.

Alas, an issue as critical in pervasive computing and services based on wearable devices as power consumption is unmanaged in RPC by any means; due to the facts that a) RPC works with a request-answer model employing potentially numerous request-answer transactions and b) Delay in RPC is about 100 milliseconds long [Wei 11], it is sure that RPC will consume much of the power a wearable device or a node in a ubiquitous network are able to store, severely limiting the amount of time they can be operational.

Remote Method Indication (RMI) switches to a more object-oriented perspective, but essentially it is a Java implementation of RPC, with the corollary of hampering any effort to make it work in a more flexible way. Additionally, RMI has the problem of providing a Java-only codification, so additional efforts in porting the code to other programming languages would be needed if wearable devices and/or nodes were alien to Oracle manufacturer.

Finally, Common Object Request Broker Architecture (CORBA) is strong in allowing a plethora of hardware (multiplicity of platforms) and software (multiplicity of programming languages) products to interact in a more or less seamless manner. This would be of an interest in the context of the Internet of Things and Ubiquitous Computing; unfortunately, CORBA will simply exceed the computing limits in most of the deployed electronic appliances in an ubiquitous environment [Roma&00], making it unhelpful for the purpose of services provided by wearable devices and, by proxy, mirroring many of the issues that have been found when trying to use the other middleware implementations. Nevertheless, the holistic approach that CORBA takes is actually inspiring, and it will not be rare to learn about ubiquitous middleware architectures that actually are a CORBA implementation hacked to its minimum, less resource-demanding features.

Considering all the trouble that discourages the use of unmodified middleware solutions already developed for traditional networks, it becomes clear that new, specific middleware approaches must be attempted if they are to perform successfully under these new circumstances and constrains.

2.1.2.1 Middleware based on the family of standards ISO/IEEE 11073.

ISO/IEEE11073, also known as X.73, is a medical and health device communication set of standards that made possible the appearance of middleware that successfully carries out the duty of interconnecting devices related to the field of Health Informatics, usually named Personal Health Devices (PHDs) [Yao&05]. Since these PHDs are often wearable and wireless by its own nature (badges, patches, etc.) they become potentially part of a ubiquitous environment, although the family of functionalities belonging to this standard are strictly encased on the medical hardware, therefore crippling its usefulness for any other kind of services. This family is strongly based on the standard IEEE 1073, which has been around in industry since the 1980s [24X7 07].

The main objectives that motivated the development of this suite were the need of interchanging data about a final user –or more specifically, a patient- by using standardized interfaces, thus avoiding the trouble that a lack of a common, established implementation may take along (communications hindered by dissimilar working equipment from varied vendors, forced bonding with a small number of manufacturers, etc.) while keeping three key features:

- **Real-time**. The monitoring and event-driven actions can take place at every moment, a reasonable goal judging from the sort of final user that will benefit from the services gained via a wearable device. This feature involves all the data harvesting, compiling and transferring from all the PHDs that while worn by the patient, may or may not be composing a PAN on their own.

- **Plug and play**. In accordance with the idea of merging the wearable computers within the environment they are placed, PHDs have to be managed by the patient or the staff of paramedics the least possible. Thus, the only action that paramedics should care about is plugging and connecting physically all the required appliances, leaving the more logical tasks (detection, configuration and communication) to be tackled by the devices on their own.

- **Efficient exchange of data**. This implies that intercommunication must be executed with the least cost possible, mostly in terms of energy usage and speed. We already know that slow and power over-consuming systems do not stand a chance in a ubiquitous environment.

As an overview of the family of standards, it can be stated that they will work under the principle of Agent/Manager Application Processes, in a way, resembling client-server architecture: the agent will usually provide all the data collected from the wearable devices, and the manager will carry out some other tasks; the simplest of them will be storing a copy of the data delivered by the agent, but the manager will also react to update events related to the worn devices by the patient, and will even trigger events on an agent if necessary. Almost always, the manager is an electronic device, like a computer or a smart phone, which can display the data on a monitor with a format that is understandable for a human operator.

The different components of this middleware architecture are the following:

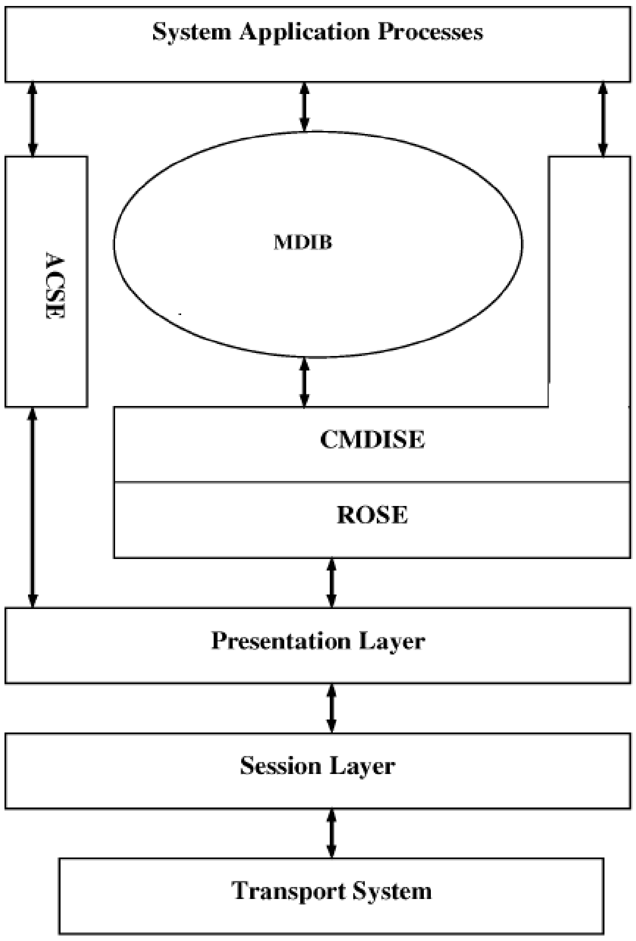


Figure 7: architecture of a ISO/IEEE 11073-based service [Yao&05].

As it is displayed, the elements belonging to the architecture are:

- **Agent/Manager Application Processes**. A part of the middleware not defined by this set of standards which is used as the interface between the protocol of the local machine and the ISO/IEEE object environment.

- **MDIB (Medical Data Information Base).** An object oriented database where the objects typical of this architecture (MMOs, Managed Medical Object) are kept safe in a hierarchical tree-like distribution called Domain Information Model or DIM, which is of great importance because it defines the storage of the objects containing the information of the vital signals. Besides, in order to ease the access to the data structure, it is split into different areas of content named packages: Medical package, Alert package, System package, Control package, Extended Services package, Communication package, Archival package and Patient package.

- **ACSE (Association Service Control Element).** This element is an implementation of standards ISO/IEC 15953 and ISO/IEC 15954, and is responsible for the establishment and disestablishment of connections, with no further data interchanged.

- **CMDISE (Common Medical Device Information Service Element) and ROSE (Remote Operation Service Element).** These elements are created in order to exchange MMOs. Services provided by means of CRUD operations such as CREATE, UPDATE or DELETE create, update or delete this MMOs. Reports are used to activate different operations on the agent or manager side. CMDISE is complemented by ROSE for services that usually imply remote operations.

- **Communication System (Presentation layer, Session layer, Transport system).** All the layers below the middleware that unable communications, with its local particularities.

Overall, family of standards ISO/IEEE 11073 can be used when implementing four different kinds of features in a service provided in a ubiquitous environment, each one treated by a different specialization of the standards: device data, general application services, Internetworking and gateway and transports (cable, wireless).

Nonetheless, in spite of its solid structure and careful design, standard ISO/IEEE 11073 has enjoyed a very limited success. Among the leading manufacturers of wearable PHDs, only Philips has taken a limited interest in it (and it has to be considered that Philips manufactures many other electronic appliances, unrelated to ubiquitous computing). Several reasons for this have been quoted: this set of standards may have a too complex structure and, perhaps more significantly, ISO/IEEE 11073 is a standard preceding all the development of ubiquitous health devices, rather than following it. This is a good idea in order to offer a framework for brand new developments, but in this case it may ignore the incipient ubiquitous products that are being made and alienate manufacturers from that standard.

As a consequence, although creating a predefined pattern for applications to be developed is quite useful and foresight-like, it has had no previous usage in the industry, and vendors are more likely to stick to their own proprietary, trusted and already deployed solutions until a standard appears and becomes dominant [24X7 07].

2.1.2.2 Aura.

According to its creators, the main goal of the Aura project, which was developed in 2002, is keeping services available for the user no matter where they are currently placed, even and especially if that means reconfiguring dynamically a task already interacting with a human user. The first glimpses of this architecture appeared while conducting research activities driven to create some sort of middleware architecture capable of a successful performance in a ubiquitous computing environment [Aura 02].

Having chosen Aura as the name of choice is not devoid of meaning: this middleware architecture will create an area around the user (the *Aura*) wherein information and computing services will become persistent, in spite of their location, and therefore executable at any place. Obviously, as there are many different areas in a ubiquitous environment, with electronic devices of potentially very different nature and performance, if the current and running services are desired to be kept accessible, the *Aura* will have to adapt to the available resources of the environment, thus moving around the representation of the task, thus adding a self-tuning feature to this middleware architecture. What is more, Aura also aims to anticipate to the actions of the user, enabling service prevision to a degree.

There are several specific components in this middleware architecture [Campo 04], which are:

1. **Intelligent networking** at the lowest level, where depending on the duties from a holistic point of view (the service the ubiquitous network is allegedly suppose to offer), monitoring data transfer from one node to another one will be taking place.

2. **Linux kernel**, governing a particular piece of hardware upon the whole architecture has been mounted and programmed.

3. **Coda**, a file access system that will allow the usage of the supplied contents on different modes, depending on the bandwidth available and the mobility of the device that made the request in the first place, with an added offline option.

4. **Odyssey**, in which the main tasks to be performed are resource management and ensuring support to context adaptation for applications.

5. **Spectra**, a component providing a remote execution mechanism that will make use of user context so that the best way to deal with a remote petition will be chosen.

6. **Prism**, a task manager acting as the embodiment of the *Aura* concept. Its lead functionalities are capture and management of user intentions, and in order to achieve them there are several sub-modules aiding Prism: a context observer, an environment manager, and the abstract service suppliers –since merging all these services will compose a task dealt by Prism-. When a task provided by the abstract service suppliers is encased in an environment where there is a determinate working context for a user, the user intention can be inferred with ease (for example, if a logging in a computer –the task- at the office –the environment- on a working day in the morning with a valid username –the context- has to be done, it is clear what a worker who has been assigned the username has planned to do for the next minutes, regarding the computer, when they arrive at the office)

In addition to all these components, as the expectable limitations of this ubiquitous environment have been taken into consideration, another feature is put to a use here: the “cyber foreign” usage of hardware appliances, such as a Personal Computer, embedded in the system, to delegate the more resource-demanding (both in terms of computational and energy power) software-related activities to other devices that will not overload the network or the nodes. Once the embedded appliance has obtained the needed output, it will be either stored or retransmitted to the ubiquitous Wireless Sensor Network.

A global vision of Aura middleware architecture is represented right at the start of the next page:

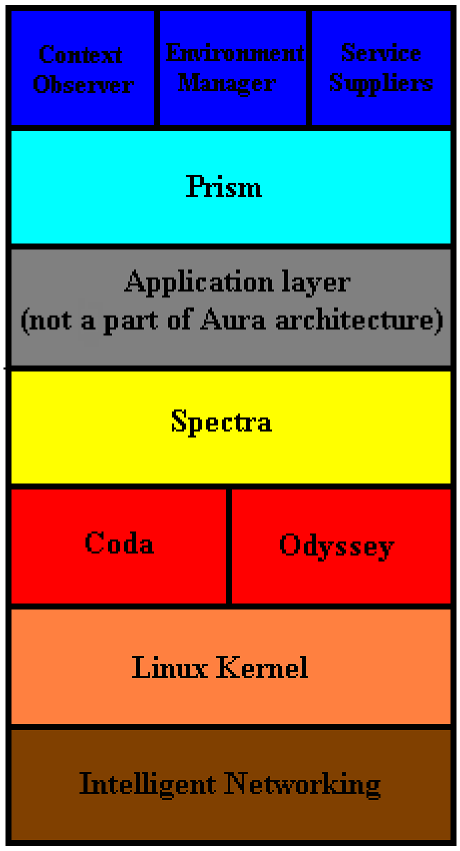


Figure 8: disposition of the components of Aura middleware architecture.

Finally, it should be remarked that Aura is capable of implementing some security services which are dependent on location user information, along with all the other functionalities [Txarramendieta 06]. The idea behind these is that security will be provided in a specific moment and place, to a specific person, in a specific level. Thus, groups of users will be created to gather all the people with common security features.

2.1.2.3 Gaia.

This middleware architecture appeared in its current form in 2005 [Kjaer 07]. It shares many of the concepts conceived by Aura architecture, especially the ones related to the challenge of adapting data and applications to the possibilities of the current devices that are providing them. For example, where Aura made use of an *Aura* to name the place in space where all the services would be stored and interfaced, Gaia is implementing a “Smart Space” where once it is active –turning into what is called an Active Space-, the applications and services will be dynamically mapped according to the available resources of the device that uses them, as it was done when using Aura. An Active Space is composed of a cluster of devices that have a strong relation with the final user and all their software capabilities and resources available, from operating systems to applications.

This middleware approach makes an extensive use of ontologies and context awareness [Hung&04]. An ontology can be defined as a structure that organizes into a standardized hierarchy a collection of concepts referring to elements bound to a domain, defining relations among them and offering the possibility to reason and infer behaviours between these entities of the domain. Among the available ontology languages, Gaia makes use of DAML+OIL (that is, a combination of DAML, DARPA Agent Markup Language and OIL, Ontology Inference Layer) to generate ontologies, although this language has been superseded currently by OWL (Ontology Web Language), which will be discussed later.

The most remarkable components of this Active Space which is at the core of the middleware proposal are the following:

1. A hardware devices collection or a **hardware cluster** bound to greater or a lesser extent to the final user. Depending on the scope of the usage of this middleware approach, the cluster will be integrated by regular-sized electronic appliances (PCs, printers, etc.), or oriented towards a Body Area Network. As usual, the nature of the hardware components can be expected to be fairly heterogeneous. Should the hardware cluster have a greater scope than a person, a human hardware administrator may be needed.

2. A **collection of software components**.

3. A **context provided** in varied ways: either by an external device or by a context observer (in a not dissimilar way to Aura´s).

4. A **set of applications** depending on the context provided and the particular devices from the hardware cluster. As far as this approach is concerned, applications are little more than context consumers, given that applications are supposed context-aware. Often, an application framework is available to develop brand new applications for the hardware cluster.

A figure with all the features related to Gaia´s active space is shown on the next page:

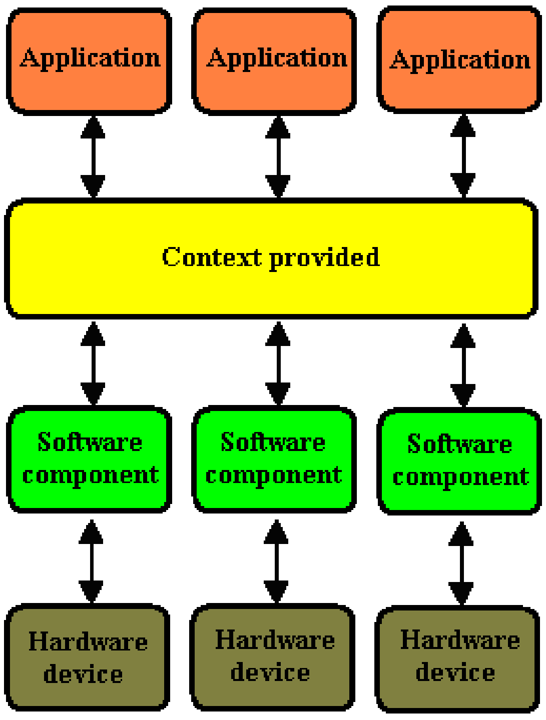


Figure 9: components in the Active space of the Gaia Architecture.

2.1.2.4 Oxygen.

This middleware ubiquitous architecture was started to be developed in 2001 by Larry Rudolph with the idea of providing a human-centric middleware platform where all the interactions were possible by using the “natural” perceptual human interfaces (speech and vision), rather than typical computer interfaces, such as keyboards or point-and-clicking gadgets [Rudolph 01]. It is an example how even more than ten years ago there was a very specific vision of what a ubiquitous system should be: a system that blends computation into people´s common life in a quiet way, by just elongating the usual features of a human being.

Rather than using a set of new components, Oxygen makes use of many popular existing elements which are in wide use, aiming to integrate a number of technologies in; particularly, the Oxygen platform distinguishes three core components within its architecture:

- A **handheld device**, which is supposed to have the common functionalities of a mobile phone: cellular-type connection, wireless Internet connection, pager, radio and a music-and-video player/recorder. According to Larry Rudolph, it is referred to as H21. The idea is that this device will be the hub from where the final user will make all its requests and interactions.

- A **network**, capable of supplying user data easily as long as the user is willing to allow it. In order to guarantee that, there are two different subsystems used: a self-certifying file system (SFS) and a location system called Cricket. SFS will deal with key management and file security separately, thereby making possible sharing the file system without intrusion inside the user domain by alterations in their private keys. Cricket, on the other hand, will have several beacons in the wireless interface emitting radiofrequency and infrared signals that, unlike other location systems, will not request location information, but will broadcast information that can be received by the elements present in the network. The key difference in this case is that it will be that a final user device will be aware of its location, instead of the system, giving the final user the option of revealing their location or not. Oxygen documentation names this subsystem as N21.

- A **sensor-rich environment** capable of receiving information from either H21 or N21 and take actions consequently from the collected data so as to regulate ambient parameters. It is named E21 by Rudolph.

An example of how the layout of an Oxygen deployment with all its expected components would look like can be observed in the next figure:

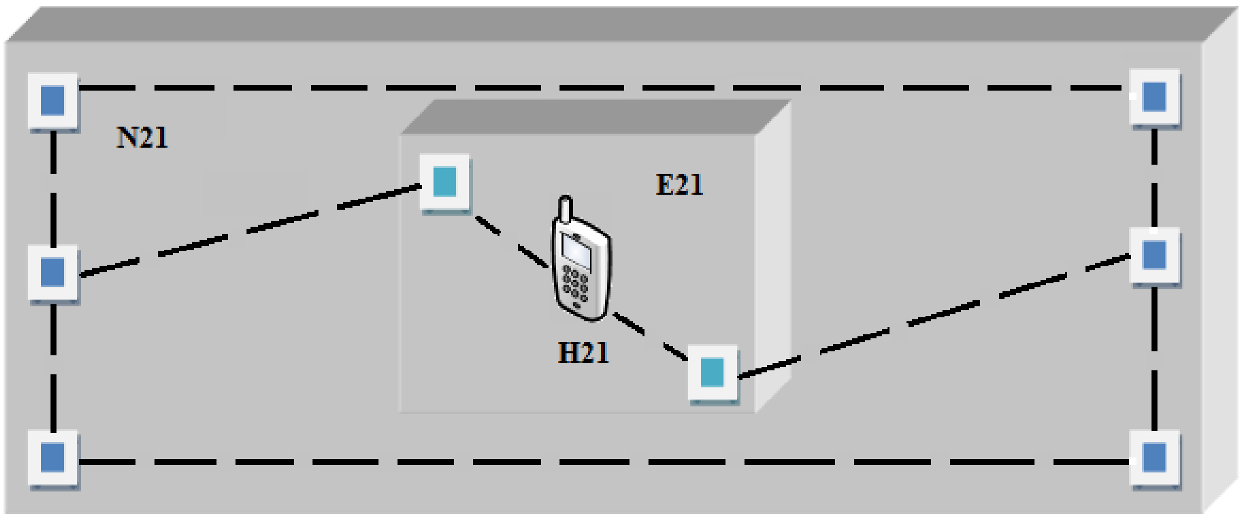


Figure 10: an example of an Oxygen scenario.

Once the elements of the scenario have already been installed, they will interact with each other according to their respective roles. Oxygen tries to implement some security measures: one of its objectives is creating the simplest possible human-machine interface in order to make the devices easier to use and, therefore, more secure. It also tries to put an effort in achieving the communication needed by the human input interfaces referenced before (vision and speech) by suggesting a set of appliances to be installed –notably, an array of microphones and cameras for speech processing-, while output interfaces will be more conventional ones –namely, large displays-.

In addition to that, Oxygen suggests a communication-oriented language and a built-in middleware [Rudolph 01b]. Although Java could be easily used as the programming language since all the devices used can offer Java interfaces, the idea of this communication language is that it will be trimmed to its unavoidable mandatory tasks, distinguishing four different kinds of components: nodes (anything that can use a name and communicate via sockets), edges (a directed connection between two nodes), messages (the entities interchanged in an edge connection), and actions (a pair trigger-consequence, where the trigger is an event such as the creation or destruction of a node, and the consequence is another event activated by the trigger, such as deactivating some nodes or establishing new edges). As for the middleware, it will be the entity that will execute the code uploaded on the different elements of the architecture. The intention of Larry Rudolph was to create a decentralized, fault-tolerant middleware that interacted with the system by performing events that would trigger actions.

An example of how the whole system would work is offered by Rudolph himself: when dealing with a computer-mediated seminar presentation, several actions will be carried out, where the different parts of the system will get involved:

1. When the person who is to carry out the seminar presentation (hereinafter, the presenter) enters the room, handheld H21 should communicate with the environment E21 embedded in the network N21 so as the presenter is acknowledged by the environment. This E21 environment will have all the nodes required to materialize and manage the presentation (camera, LCD or laptops with the intelligence and the electronics capable of becoming part of the environment).

2. Once E21 has recognized the presenter, human interfaces as speech will make possible that the presenter operates with the presentation (moving slides forward, backward, jumping from one to the other, adding notes) by using one command chosen from a small collection (“next”, “back”, “go to 13”). This can be used in order to assume the functionalities that would require a wireless mouse or a laser pointer, too.

3. The audience may want to watch the files the presenter is using (slides, documents) in their own laptops or mobile phones. Thus, they may have to sign in E21 or, if they were previously on N21, N21 itself will take care of that operation.

4. In case that any other document useful for the presenter that is not part of the current slides is required, they should be accessed without giving away their placement (in the mobile of the presenter rather than in their laptop, etc.) as the signing and acknowledgement of the process gave the system access to any available repository, provided that there are no security issues.

All in all, the Oxygen goal of merging already developed technologies and appliances into a system in compliance with the paradigm of the Internet of Things is enriching as a sandbox, where existing elements integrate with each other, but it offers only a half-hearted ubiquitous ambition, as the only added value of this middleware approach is the integration of the components instead of innovating with new ones, either of hardware or software nature. Plus, its focus on the technologies of the moment can turn the architecture into an obsolete one as soon as a new, incompatible technology turns up.

2.1.2.5 One.world.

This middleware architecture appeared first in 2004. It was launched by Professor Robert Grimm, from the New York University [Grimm 04]. Its main target was creating a framework for the development of pervasive applications for buildings, with a stress on self-adapting the rapidly changing conditions of a ubiquitous network. As it happens with the other architectures, Grimm tries to accomplish the idea of creating an environment where all the data can be accessed seamlessly throughout any room of a building (he uses a digital biology laboratory as an example) without any need of configuring portable devices -such as laptops or mobile phones to access wireless networks-, or to log in and out, closing and restarting applications and, in a nutshell, explicitly interacting with the carried electronic device to do any operation different than retrieving or consulting data.

One.world tries to cope with typical issues related to ubiquitous and pervasive computing that have been self-named and trimmed to three: the change of the context as the user moves by, ad hoc connections among devices and the readiness to share information. When dealing with them, this architecture will find support in the four pillars that make it possible:

1. **A virtual machine**. Since all the code of the One.world architecture runs on the Java Virtual Machine (JVM), the existence of a virtual machine is mandatory. In addition to that, it is a feature that turns to be quite useful if the unpredictable heterogeneity of the different devices that may take part in the ubiquitous architecture is considered: a virtual machine can hamper the differences among devices and provide a homogenous foundation for higher levels, as if it was a middleware by its own.
2. **Tuples**, in a technical context, a tuple is an ordered list of elements used as a mean of data representation [English 11]. In this case, tuples are self-describing records with named and (optionally) typed fields; they will be used to store data under a common data model, including a type system as well, for all the applications available.
3. **Asynchronous events**. They are the vehicle used to notify any change in the runtime context to the applications.
4. **Environments.** They are what processes are to operating systems: the element that runs the host applications which is also isolating them from one another. Furthermore, they store persistent data that, since it is likely to be used by several different applications, can easily generate application clusters based on persistent data sharing. Finally, environments can be nested within wider environments, allowing the “super-environment” to modify the course of a whole “sub-environment”, according to the hierarchy.

Another central concern for the creators of this architecture is what the application requires in order to successfully perform its duties. The actions required, as well as their implementation in the scenario of One.world are as follows:

Application requirement	One.world service
Communicate	Remote events
Fault-protect	Check-pointing
Locate	Discovery
Move	Migration
Search	Query engine
Store data	Structured I/O

Figure 11: requirements for an application running on One.world and mapped services.

According to Robert Grimm, the most compelling services of the architecture are Discovery and Migration. The Discovery service will get resources –that is, event handlers- by paying attention to their descriptions. It will support a set of options, including early and late binding, unicast and multicast communications. By definition, Discovery service must be self-managing, as it will have to adapt almost constantly to shifty context environmental circumstances in order to be efficient enough. Migration service, on the other hand, will essentially copy an environment –that is, including all its components- to another mobile device, in this way reducing the effort to port an already working application to a brand new mobile machine, and facilitating its use in a context where the user has an appreciable mobility.

When dealing with the issues presented before (change of context, ad hoc connections, information sharing), several parameters will be considered: completeness (capability to build complete programs), complexity (hardness to program application with this middleware architecture below), performance (ability to support system loads) and utility (usefulness to build pervasive applications using One.world as middleware architecture). This middleware architecture was tested employing several services specifically created for One.world: a replication service (to gain access to ubiquitous information), a user-application manager (to have the user handle and manage an application on their own like, for example, transferring from one device to another), a text and audio messaging system and a graphic laboratory project called Labscape. Regardless of their very different functionalities, some underlying conclusions could be extracted:

- The applications tend to use nesting environments in order to have the applications working in several different devices (as when the replicated data are accessed or the applications are managed by the users).

- The applications can be accessed in almost every device present in the environment, in accordance to the paradigms this middleware architecture is based on.

- Mobility of the user is not a threat for them; in fact, it is encouraged as a way to better interchange data and information context from one environment to the other.

Despite the approach and the implementation of measures closely related to the paradigms that will be reviewed later, like Service Oriented Computing and The Internet of Things, this architecture has some limitations, as Grimm himself pointed out. The problem at its core is that One.world puts its efforts in the Java code generated in the programming stage, rather than in the data that is interchanged –One.world is conceived as a “code-centric” middleware architecture, not as a “service-centric” or “used-centric” one-, which can be a problem when the system is confronted with any non-Java coded component, forcing to do some porting in order not to spread that problem to higher layers. To counter this issue, Grimm suggested the change to a “data-centric” perspective, where data would be interchanged by using already existing standards centred in data, such as XML Schemas, so that the elements interchanged are not Java code but metadata.

In addition to that, One.world is using its own protocols for the wireless communications among nodes. Although it is comprehensible because of the technological constrains that are found in this sort of environments, Grimm admits that its use should be discouraged, as nonstandard protocols will become a source of nuisance when the architecture is expected to integrate with other distributed systems (most notably, Web-based applications) outside One.world.

2.1.2.6 PCOM/BASE.

PCOM (acronym created after Component System for Pervasive Computing) was first introduced in 2004 by Christian Becker et al. at the University of Stuttgart [Becker&04]. The main point of view behind this architecture is that the components in a pervasive, ubiquitous environment, when interacting to each other, reveal a series of dependencies satisfied by other components when making those very interactions. These relations of dependency can be considered as *contracts*. Therefore, distributed applications can be specified in this context as long as they are made up of inner components designed explicitly with dependencies modelled using contracts as a template.

Apart from this middleware ubiquitous architecture, PCOM will rely on an integrated, flexible communication middleware named BASE, which has been programmed in J2ME and makes use of the Connected Limited Device Configuration (CLDC). BASE provides an important support at the communication level by dynamically choosing among the different available communication protocol stacks the best possible one, even for a communication that is taking place at that precise moment. What is more, it will offer the application developers dynamic mechanisms to ensure device discovery and service registration, which are critical in a ubiquitous scenario –where the amount and kind of the devices present is often fluctuating due to crashing of nodes, energy shortages, installation of new elements without notice, building of brand new services on the spot, etc.-. Finally, BASE is structured as an extensible micro-broker that will adapt itself to the allocated resources of the network, seeking the goal of being able to run on devices with poor resources and taking advantage of the devices rich on them, offering generic, automatic adaptation support at the communication layer.

On the other hand, the role of supporting adaptation for an application at a higher stage (typically, application layer) will be undertaken by the PCOM architecture throughout the study of three possible solutions:

- Manual adaptation lead by the final user (the user is offered several options and it is up to them to choose what suits better for them). This option is criticized for being against the principles of pervasive computing, as it requires the final user make an interaction and the application cannot do without it. What is more, since in a ubiquitous environment services may become rapidly unavailable, user input may be needed at an unacceptable pace.

- Application-specific automatic adaptation, where the duties related to adaptation are tackled by the application itself; typically, the application will signal changes in the environment and will reorganize its parameters according to them. This option, although attractive on paper, is resource-costly and, according to Becker is *"[…] leading to complex and error-prone adaptation routines".*

- Generic automatic adaptation, which can be considered as a middle way between the other options. Parameters for an input and/or output service will be specified by the programmers and while in runtime, the system will try to find service with a good enough quality. This is the option that will be used by PCOM to ensure adaptation support at the application layer, while BASE will be using it as well for the communication layer. BASE will also be the host for the PCOM components that makes PCOM´s runtime system possible.

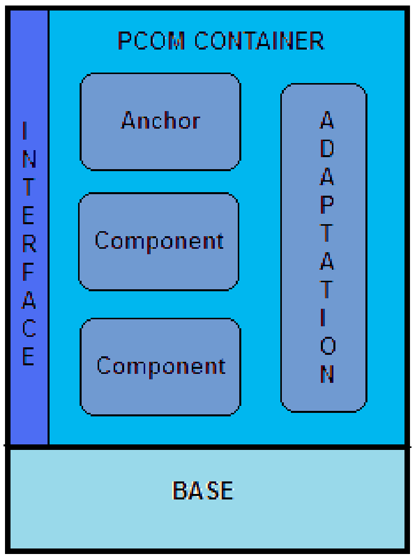


Figure 12: general architecture of PCOM/BASE.

As it can be seen in the previous picture, there is a hierarchy with several elements of the architecture. The most relevant are two:

- **PCOM Container**, where sub-elements are encased, such as a) A container interface in order to operate with other PCOM containers, b) Several components used to inter-articulate into an application, c) An anchor, which is no more than a component but since it is the root one will have some additional features (it will identify the resulting application) and d) An adaptation element that will carry out the necessary duties to either signal any alteration in the availability and quality on notified components throughout the network, or any user strategies already programmed to deal with the changing components.

- **BASE middleware**, a critical part of the system as it is responsible for the communications at the network level and it also select services among the present ones.

The components in the network work providing services determined and standardized in this environment by the usage of contracts, as it has been learnt before. Also, these components perform their tasks in a component lifecycle defined by the PCOM container, consisting of two different states, STARTED and STOPPED. In this way, the container will load the representation of a contract that, at the same time, is representing the features and requirements of an object. If this component is able to fulfil the needed requirements, then the component will be embedded into an application, and when all the required dependencies are fulfilled, the component will change from STOPPED to STARTED state. At this point, the loaded component will provide its functionalities until they are either depleted or no longer need, therefore reverting to the STOPPED state until the moment this component may be required again.

Service selection, on the other hand, will be made both by the PCOM container and BASE, although PCOM, as it is at a higher level than BASE, will select services via components. This fact has prevailing consequences: components add more flexibility and adaptation than the simple services, but they will introduce a delay and will have a slightly lower performance when they are compared with BASE results, although BASE is not operating at the application layer. That is why, despite the huge effort made in solving the main challenges of a ubiquitous environment and its applications made by this architecture, and providing an initial solution for mobility, update and reliability issues, it somewhat blurs those achievements by introducing some redundancy with inconclusive usability: the idea of having two layers competing to solve the same task is not consistent with the concept of a layered architecture, where the lower layers provide functionalities to the higher ones, and in addition to that, additional costs in performance -adding a small delay in PCOM service reselection- and resources -according to the scientific paper composed by Christian Becker et al., the inclusion of PCOM container asks for additional 30-40 Kb of memory on top of the 90-120 Kb already required by BASE, one more reason for not having two separated layers performing similar tasks sometimes-.

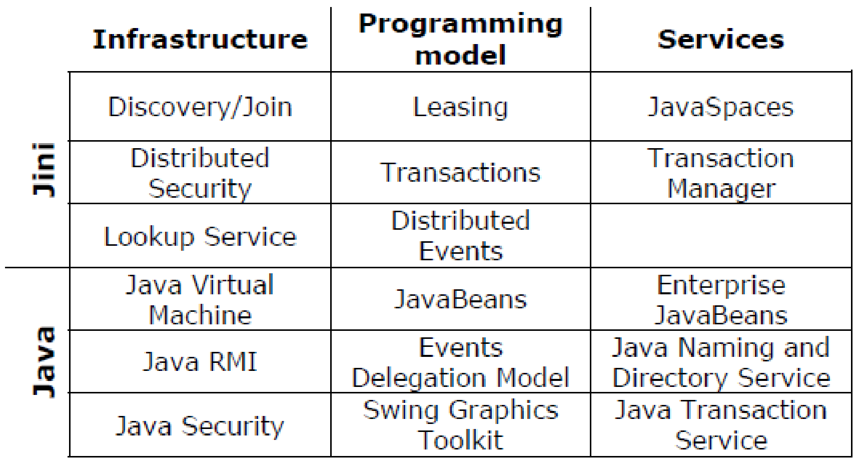
2.1.2.7 Jini.

Jini was put forward as a model for ubiquitous middleware architectures by S. Ilango Kumaran in 2005. The goal of Jini is providing a middleware that will cope with all the troubles of having different hardware, protocols, drivers or operating systems using a principal called “spontaneous networking”. Due to its design, components or nodes within this model do not need to acknowledge the existence of any other element, being the only prerequisite that the participants have a functional Java Virtual Machine mounted as part of them. In order to make any non-Java-based component usable by the community, they will be wrapped in Java code, in an attempt to make the system application-centric rather than language-centric.

One of the core concepts of Jini architecture is the *lookup service*. They are self-managing Jini services that, unlike all the others, have governing functionalities within the architecture. Lookup services will organize the published services as well as answering requests done by clients, and references to them will be obtained by either multi or unicast Jini communication protocols.

Another one of the core principles of Jini is the idea of “leasing a service” to the community of devices present in a Jini deployment. When a service provider wants to register a service in the community, it will begin a service registration process with a lookup service. In order to complete it, the lookup service will grant a lease to the service provider regarding that particular service. This lease must be renewed – lookup services make multicast announcements of their presence from time to time in other to ensure this-, or else the service will be de-registered. In addition to that, clients and servers in the Jini community can register for events related in a way or another in one lookup service (as new or discarded lookup services); clients can even subscribe to events related with functional, modified or discarded services. All these event registrations are also leased into the federation, so they can be used as a tool for updating the service layout of the system.

A chart with the components of a Jini architecture is below [Hariha 05]:



	Infrastructure	Programming model	Services
Jini	Discovery/Join	Leasing	JavaSpaces
	Distributed Security	Transactions	Transaction Manager
	Lookup Service	Distributed Events	
Java	Java Virtual Machine	JavaBeans	Enterprise JavaBeans
	Java RMI	Events Delegation Model	Java Naming and Directory Service
	Java Security	Swing Graphics Toolkit	Java Transaction Service

Chart 1: Jini architecture and comparison with Java functionalities.

As far as the Jini architecture is concerned, there are three main components:

- **Infrastructure**, built on Java´s Remote Method Invocation (RMI), despite its poor capabilities for this sort of context. It provides the discovery and joint protocols used to federate the potential elements of the architecture, whether they are hardware of software-based. The discovery protocol will come in handy when clients or servers are searching a lookup service – which eventually will answer with a TCP-based unicast response to their requests-, and the join protocol. Besides, as in fact Jini is based on a network of Java Virtual Machines, all the functionalities of a Java Security environment can be put to a use in a distributed environment, thus having Distributed Security all throughout the architecture. Finally, the core concept of a lookup service will enable a client interested in joining a Jini federation to find present services and make sure of their usability, not in a dissimilar way to CORBA or DCOM, which are distributed but not ubiquitous-based.

- **Programming Model**, with the idea of extending to a distributed and ubiquitous environment Java elements such as JavaBeans, an Events Delegation model and a Swing Graphics Toolkit for graphics generation. The other core concept mentioned before, service leasing, is used not only as one of the main functionalities of the system in order to enforce the normal behaviour of the Jini architecture, but also as a mechanism of self-healing and updating after a partial network crash –which are much more common in environments with the scope this Final Degree project is disserting about than total and complete network failures- as it will multicast itself with all of its services under different conditions –functional, outdated, etc.-. Plus, transactions will have to be employed as well. Jini defines a transaction model where there are two stages needed for completing a transaction but does not give any detail on how to implement them; in a way, leasing can be considered a transaction mechanism as well. Thirdly, a Distributed Events model, based on a Java Event Model in JavaBeans will support event remote registration and propagation by using a distributed event protocol that will let know any object in the federation if there have been any change in a singular service or even in another object.

- **Services**. The approach intended to be taken by Jini is that services –and by proxy, the applications that provide them- are the main focus or the architecture, with the pretension of regarding everything as if it was a service. One more time, lookup services are used in a recursive way to find services that will offer different functionalities according to the needs of the registered object.

Jini tries to use the idea of service and application-centric architecture to its advantage: it will usually give the same treatment to hardware and software components, as in a pervasive environment, it is the service they provide what Jini is interested in. As for mobile devices, since sometimes they will be too resource-constrained, Jini provides some initiatives (Jini Surrogate Architecture for networking, ServiceUI for interaction in pervasive environments or even a Jini Mobile Edition for a easier handling of codification tasks) to accommodate its utility.

In theory, Jini offers a series of concepts very appealing: its view of the architecture as a service-centric network, or a distributed transaction model as a way to communicate the elements among them are in harmony with the principles of the novel tendencies in wearable computing. Unfortunately, it often feels as if they are mostly a theoretical pretension: sometimes the line separating an application-centric architecture and a service-centric architecture is used in a diffuse manner, and it can have serious consequences in the performance of a Wireless Sensor Network or any other component of a ubiquitous environment (as it is not the same making the centre of the architecture applications, that are usually useful for final clients, than services, that can be used for internal purposes, as it is proved by our own semantic middleware architecture, nSOM). Moreover, although objects not based in Java implementations can be wrapped into one codified in Java, it is not explicitly mentioned, nor there is an implementation of it, how it is done within a Jini context. Finally, the idea of having to take into account computing capabilities of the devices involved in a Jini system is not consistent with the idea of abstracting hardware and/or software components in a pervasive environment.

2.1.2.8 UPnP.

UPnP (which stands for Universal Plug and Play) is a Microsoft architecture first issued in 2006 in order to make its own contribution in pervasive, distributed environments. Initially conceived as an extension of the plug and play concept into a networked, wireless environment, it ensures an unusual degree of openness and independence: UPnP is an open source, open architecture which does not depend on platforms or languages for its right performance, while following and making use of some of the best known standards of industry, such as HTTP, SOAP and XML. The target context for UPnP is mostly either a home or a SME environment, where all the devices belonging to the building they are present are interconnected by means of this architecture [UPnP 06]. UPnP intends to integrate all the devices found in these scenarios seamlessly, without making any intervention from the user necessary (if so, in a plug-and-play interaction fashion).

There are three main components behind UPnP [Intel 11] which characterize the architecture:

- **Devices**, entities capable of establishing connections among other devices by making use of the UPnP-defined communication protocol. Despite their name, they can be either of hardware or software nature.

- **Services**; usually offered by the devices, they are actions that can perform duties that will result in a benefit for the final user. Since UPnP is taking part in a consortium constituted by the industry (the UPnP implementers Corporation), some of this services are already fixed and defined for some of the devices typical of UPnP, as they are already existing in the market place. Services will usually provide one or more actions related to their functionalities.

- **Control points**, an element of the architecture that will extensively manage the relations between the devices and the requests done towards their services. Control points will discover and invoke service actions associated to devices, and are able to subscribe to events and actions provided by services.

A representation of an effective data interchange under UPnP requirements is as follows:

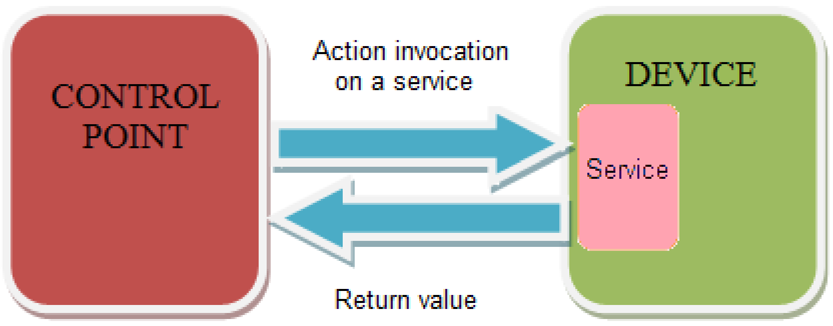


Figure 13: interactions between Control Point and device elements of UPnP.

The depicted process can be explained like this: a) A device connects itself to a UPnP network, thus starting a discovery process aimed to get to know which other devices and services are present in the network. For this discovery process to take place b) A control point will perform queries to the network using Simple Service Discovery Protocol (SSDP), identifying devices and their hosting services. Therefore, c) A device offering a service will send an answer to the control point with a URL containing the device description and the service that it can offer. This will be a file written in XML that can be looked upon so as to obtain information to understand facts as the make or the services offered by the device –for example, input and output methods used by services-. In the end, d) The control point will make use of SOAP protocol to invoke actions on one or more services, as it may be required in the context of a request.

In a nutshell, UPnP is an architecture offering interesting concepts, and a huge effort in overcoming the stereotype of Microsoft preference for proprietary solutions and closed architectures. In addition to it, UPnP is backed by a consortium of a considerable amount of manufacturers and vendors, aiming for its acknowledgment as the architecture to use in smart home or SME environments. However, due to the fact that its only pretension is to universally connect devices, regardless of their ubiquitous, pervasive building or not, in a very specific environment, some of its features cannot be of any use for new devices and solutions complying with the novel tendencies in wearable computing, such, as semantic knowledge management or the Service Oriented Computing.

What is more, there is no mention of context awareness, which is a key concept in the scope of this Final Degree Dissertation, and adaptability is not as flexible as it should be, as it is not mentioned either how UPnP copes with failing networks nodes or service errors. Moreover, UPnP is rather focused on the devices rather than the services, which contradicts with the idea of using middleware to isolate different, even discordant, hardware architectures. Finally, the correct performance of this architecture in a fully ubiquitous environment is dubious, as it makes use of standards that are not designed for an environment under the conditions usual in ubiquitous and pervasive computing and, as it had been seen in the problem statement part of this Final Degree Dissertation, that usually renders the standard useless against these new circumstances (constrained resources, high mobility rate, dynamic adaptation to changing network conditions, etc.).

2.1.2.9 CoolTown.

This middleware architecture was elaborated and put forward in 2001 (although first efforts on it date back to 1999) by Tim Kindberg and John Barton –who would become part of the staff working at Hewlett-Packard labs, the company that would use this architecture for interconnectivity among their own devices- to address the issues and challenges that proliferated in a ubiquitous environment. The pretension of this middleware is providing an infrastructure to what their creators refer to as nomadic computing; in Tim Kindberg´s words [Kindberg 12]: "*the goal of the Cooltown project was to provide infrastructure for nomadic computing, a term the project used for human-oriented mobile and ubiquitous computing*". Under this context, this is an idea similar of more contemporary pervasive or ubiquitous concepts. Also, many of the ambitions of this architecture were shared among the other ones, namely creating a "real-world wide web" where all the existing entities of the "real" world would be mapped into a web from where the entities´ functionalities would be characterized and accessible, usually via URL – which is a simple way for providing access to the features belonging to an element present in the system -, that would serve as a web presence in this web-enabled environment.

To face the previous challenge, CoolTown makes use of a method called eSquirt for collecting links – that is, URLs- from infrared beacons attached to different devices like walls, printers or radios [Kindberg&01]. By using infrared beacons CoolTown aims to abstract itself to the kind of device they are attached to (regardless they are a printer, a RFID tag or a barcode), and implementing an URL for each device makes it accessible for any other entity in the network that has HTTP-capable devices, as it can be observed on the next page:

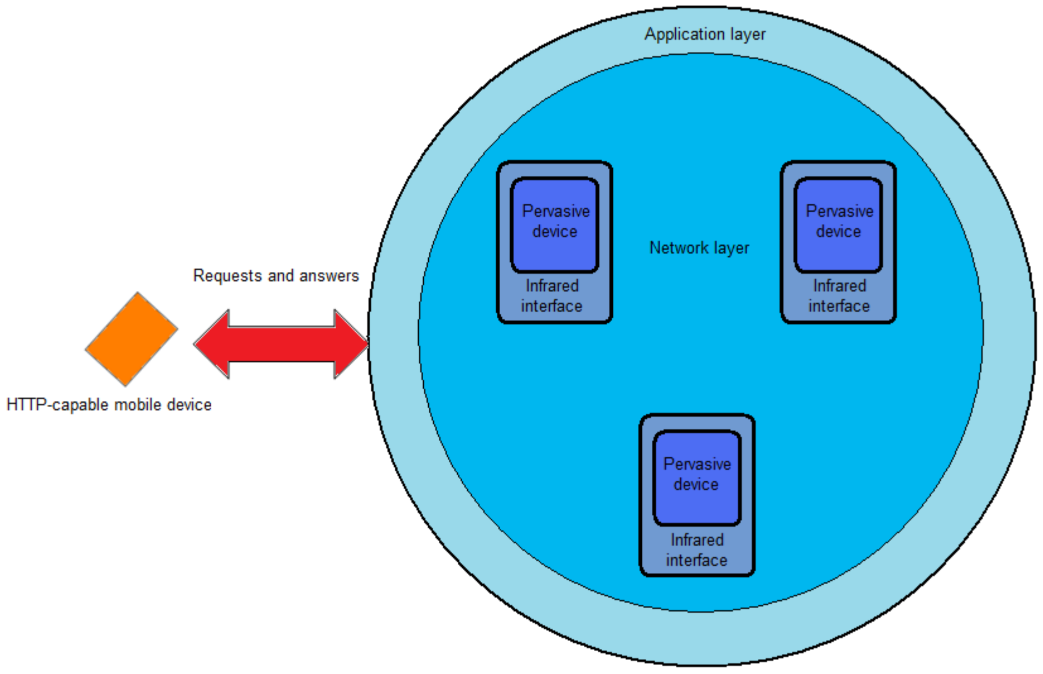


Figure 14: a general view of a CoolTown-based system.

When it is put on a use, should all the devices be present in a room, each of them counting with their own URL, as well as providing the room with one of them for characterizing purposes, a portable HTTP-capable device, such as a PDA, can request the URL belonging to the room that will be containing all the other URLs associated with the other present devices, therefore using the URL as an interface to the room. The context information will usually be offered by other web-services based on the piece of equipment that has been attached an infrared beacon. Types of context will be not only about the physical world (*where, when, who, what, how*), but also about relationships among components (*contains, isContainedIn, isNextTo, isCarriedBy*).

This middleware architecture, enriching as it is, has a concerning amount of limitations that jeopardize its usability in a ubiquitous, pervasive environment. The performance of infrared technology in this kind of wireless communications is likely to be very poor, as infrared interfaces have very limiting requirements as far as distances and obstacles go. Also, it has to be born in mind that this middleware was developed by a single manufacturer rather than a standardization organism or a consortium of several companies; it is only natural that it was focused on solving one particular problem for the manufacturer (mainly, communication between user and printing devices in a nomadic way, from a laptop, a PDA, etc.), but it does not look like it can handle all the trouble behind having ubiquitous devices belonging to different companies operating in a full-scale ubiquitous environment, with different functionalities, context data, etc. Finally, partly due to the time it started its development, CoolTown makes use of technologies alien to a wearable-oriented middleware (to begin with, IP “as is”, without any modification to ensure its functionalities in a pervasive, ubiquitous environment), which have already been discarded for the trouble and malfunctioning risks inherent to its usage.

Some of the issues present in this architecture were tackled with another Hewlett-Packard development in this same area called JetSend, which later is mentioned as other wearable-oriented middleware possible architecture.

2.1.2.10 A.U.R.A.

A.U.R.A. (acronym loosely based on application-aware and user-interaction aware energy optimization middleware framework) [Donohoo 11] is a middleware architecture whose main lying idea is saving energy and optimizing the good usage of resources (according to the main objectives of this middleware architecture) and in order to achieve this goal, every single moment that the wearable device is rendered idle,, the architecture will be put to this use. This piece of middleware is unrelated to the Aura architecture described before, so it has to be considered that they have next to nothing in common. For example, there are three kinds of processes recognized by its creators, which are consecutive and will have a user interaction with a wearable device as a consequence: perceptual (perception of an event considered as input from the environment by the user), cognitive (decisions taken by the user depending on the perceived event) and motor processes (the execution of the decision taken before). As there is some little time of inactivity at the terminal between the ending of one of these processes and the start of the other (which is named interaction slack), these small periods of time can be used to the human actor´s advantage: the CPU power and other features will not be needed, so they could be, if not stopped or switched off, at least slowed down.

As long as the middleware architecture overview is concerned, there are three components critical to its performance, which coexist with two more components with emulation and storage functionalities. The place they are located can be observed in the following picture:

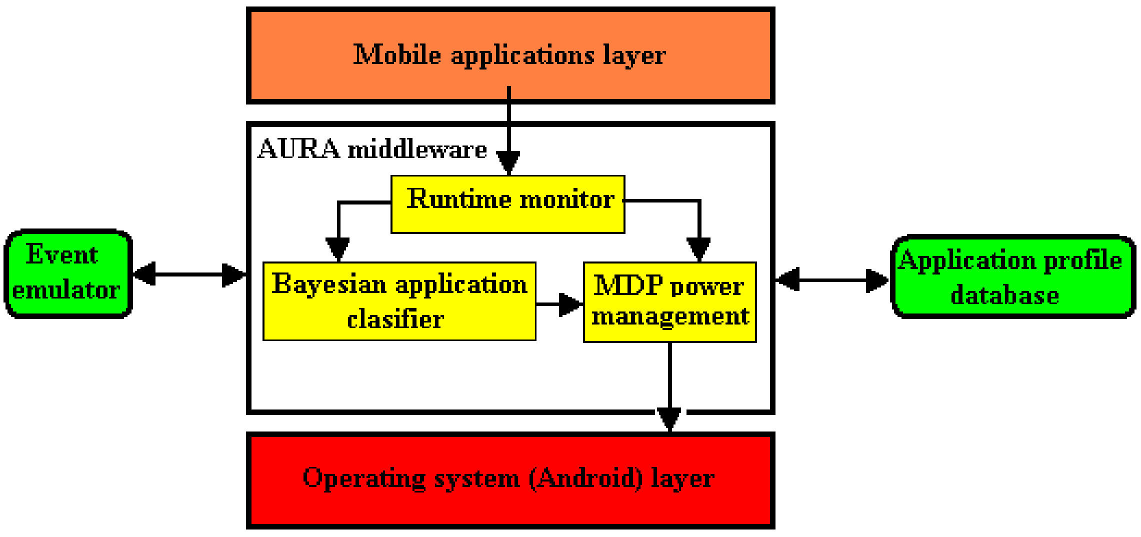

Mobile applications layer
AURA middleware
Runtime monitor
Event emulator
Bayesian application clasifier
MDP power management
Application profile database
Operating system (Android) layer

Figure 15: AURA middleware architecture and placement

As it is present on the picture, the three most notorious components of this middleware architecture are:

- **A runtime monitor** to interact between the operating system (experiments were run using Android devices) and the applications accessible via user interface. Usually, the monitor will be focused of several foreground, global applications (e.g. whether a hard keypad is used on the mobile device or not, or a particular application that is being extensively in use), either dynamically adapted to the context of the environment or behaving the same way regardless – or in spite of- it. These applications will send broadcasts whenever a triggering-application event takes place (a touch on the screen, sliding open a mobile device, etc.). Since the runtime monitor will be tracking the applications in use, the mobile state and the interaction events, if it is detected that a new application has gained foreground, the runtime monitor can easily save information from the no-longer-in-use application in the application profile database, and recover previously saved data of the new application in use, should they exist. Also, the MDP power management module can be used to the device´s advantage if the new application has already been classified.

- **A MDP (Markov Decision Process) power management module**. Its work methodology is based on the idea that if a statistic record is made of the user interactions with the mobile device, it will be inferred how the device is both the most and the least frequent way used, and thus the resources consumed by it can be managed in a more efficient way. The module establishes five parameters in order to achieve a successful performance; two of them are based on statistics, -the probability of an event going from a high to a low threshold (hereinafter, Phlt) and the probability of an event going from a low to a high threshold (hereinafter, Plht)– whereas the other three are closely related to the utilization of the CPU -Utilization from high to low threshold or Uhlt, Utilization from low to high threshold or Ulht and low-to-medium utilization or Ulmt-. Typically, when the system is used below the value of Uhlt *and* the probability of an event to happen falls below Phlt as well, the system will enter in a Below Nominal mode, where the brightness of the screen present in a mobile device is reduced and the processor frequency level switches to the lowest supported. Once the device has entered this stage, either a reception of an event or the rise of the probability of receiving an event will revert the system to its Nominal, “regular” mode. A third possibility is present if the processor surpasses the value of Ulmt: within the Below Nominal mode, the processor frequency will be sped up.

- **A Bayesian application classifier**, which will perform its duties according to the principle of Bayesian learning of using both prior outcome probabilities and present events in order to get probabilities of a future event to happen. In this module, three different levels of user interaction are defined (low, medium and high interaction) and applications will become members of each of the levels by analysing the standard deviation and the mean of the passing time between events launched as a result of user interaction.

- Furthermore, there are two more elements that should be noted: an **event emulator**, used in AURA middleware to emulate interaction events, and an **application profile database** storing all the data related to the previous foreground applications –should these data be required by an application returning to the foreground-, as well as the level of user interaction of many of them (the Bayesian application classifier will be used in case the level interaction is undefined).

To summarize, the AURA middleware architecture offers an array of options in order to improve the usage of the constrained resources of a mobile device. However, it must be highlighted that AURA has been tested with mobile devices that exceed the usual capabilities of a mote, like a mobile telephone with a popular operating system such as Android, so it could possibly become useless in a ubiquitous environment with electronic devices with lower capabilities (as they usually are). In addition to that disadvantage, AURA, unlike Aura, is strongly focused on one single task –energy saving-, and somewhat neglects many of the other functionalities typical of a ubiquitous middleware (context awareness, service management, etc.).

2.1.2.11 Other ubiquitous-oriented middleware architectures.

Besides the middleware architectures previously presented, there is an additional number of them which were either not as popular or have been regarded as nearly useless for the time being. However, given the still early state of the research and development of the Internet of Things and other ubiquitous paradigms, their contributions are noteworthy and may even be considered as forerunners for other middleware architectures.

RKF is a ubiquitous middleware implementation that appeared in 2003 as the work of Stephen S. Yau and Fariaz Karim [Yau&03]. Contrary to many other implementations, RKF is not a holistic ubiquitous middleware but a lightweight component –or as their developers name it, a *building block*- meant to be embedded as a part of a wider middleware, presumably based on RCSM (Reconfigurable Context Sensitive Middleware, developed by Stephen S. Yau and Fariaz Karim as well from 2001 on), thus providing an enhanced ubiquitous middleware with little more software and hardware requirements. Its structure is built around the idea that an interface of an object becomes a communication endpoint when associated with one of the many methods of the object. These communication endpoints (also known as IM_PAIR, for Interface and Method Pair) are not only interchanged between members of a ubiquitous network such as the wearable device or a ubiquitous node, but also are advertising the objects of the device acting as a host that are executable in the current context. In this way, IM_PAIRs will broadcast information and link components of a distributed architecture in two possible ways: either in a regular client-server mode or in a ubiquitous-like context-sensitive mode.

MARKS (Middleware Adaptability for Resource Discovery, Knowledge Usability and Self-healing) appeared in 2006 as the result of the research carried out by Moushumi Sharmin, Shameem Ahmed, and Sheikh I. Ahamed [Sharmin&06]. In their view, there were some aspects of middleware ubiquitous computing that were not being correctly addressed (knowledge usability, resource discovery or self-healing on the systems) and this misdirection could affect some of the assumed features of ubiquitous computing (Context-sensitivity, Situation-awareness, Ad hoc communications). In order to tackle this alleged weaknesses, they designed MARKS with several core components (Universal Service Access, Trust Management and a module for resource discovery) and some other interacting with the objects belonging to the application layer (a context service Module, a knowledge usability service module, a self-healing service and some free room for other service modules) which are using an Object Request Broker layer as the cornerstone of the architecture. In an experiment conducted by the designers of the middleware architecture, a number of Dell Axim pocket computers were used. These devices had Windows Mobile as their operating system, and therefore, a suite of technologies closely related to Microsoft (like, for example, Visual Studio .NET). The obtained results were good enough, and two further versions were created: a secure one, with security taken into account from its very first design, called S-MARKS, and a lighter one, called $\mu -$MARKS, conceived to be used by motes as part of a virtual wellness assistant that merges use of ubiquitous computing with e-health.

JetSend is a protocol developed by Hewlett-Packard. It was created with the purpose of having a peer-to-peerstructure and becoming a machine and media-independent communications technology, without any need of device drivers. Rather than using infrared beacons (although infrared interfaces are supported, too), JetSend employs what their developers call e-material, that is, an electronic material used for surfaces, or representationa of any job that has to be tackled (for example, if a page has to be printed, the surface will be that particular page) [Hewlett-Packard 12]. Surfaces can act as well as a shared medium for capabilities advertisement of each device present in the architecture. Again, however, although it aims at the usual electronic consumer goods that can be present in an office or a domestic environment, it does not go further than that and does not implement any functionality related to lower capability devices or context information –in addition, it still uses technologies, such as TCP/IP that do not specifically belong to the domain of wearable–oriented middleware-. Nevertheless, clearly it does not target the kind of nodes that can be found on a Wireless Sensor Network, either as part of it or as a final user intelligence-augmented common wearable, so it cannot be considered entirely as an unwanted flaw. Another development somewhat reminiscent from JetSend is TransferJet, but its utility in a ubiquitous environment or a Wireless Sensor Network is practically none, since the devices to be connected need to be either touching or being brought very closely together.

iROS application model tries to create a middleware system for interactive workspaces [Ponnekanti&03], such as any room or space technologically augmented where people gather for collaborative work. iROS consists of three different subsystems: an Event Heap for application coordination (a programming model based on entities communicating via message passing or events, with an Event Heap client library claimed to be easily portable to multiple hardware platforms and software programming languages), a Data Heap for data movement and transformation (which is used for semi-permanent data storage in an interactive workspace, regardless of being either proper data or metadata) and the ICrafter, a framework for services, or in a more accurate way, for any sort of hardware or software component that can be managed throughout the network as if it was a service.

The model put forward by iROS uses application parts which will communicate by using the Event Heap, using tuples as the structure for communication; in addition, if a requested functionality is not present at that specific moment, an aging mechanism will be implemented in order to dispose of that request. As for other middleware ubiquitous architectures, iROS seems to direct itself just for indoor, very limited environments, such as offices or rooms with a varying degree of pervasive components.

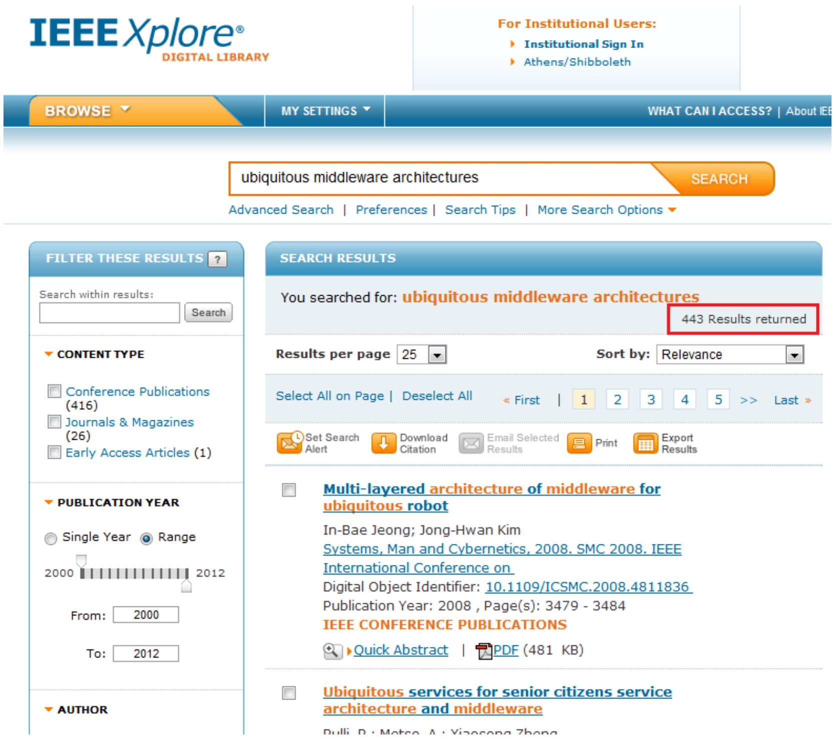
DCOM (acronym for Distributed Component Object Model) is a middleware architecture developed by Microsoft [DCOM 12] that, unlike already-presented UPnP, is a proprietary technology. Its initial purpose was to extend the functionalities of Microsoft´s COM under a distributed environment, in a way trying to solve the limitations that COM architecture had found, which were marshalling (serialization and de-serialization of return values from requests done by method calls over the wire, or more accurately, methods that have sent directly to the user the result of their request), distributed garbage collection (making sure that the references held by clients and/or applications are released when something abnormal regarding the network performance has happened, such as a crash or a client disconnection) and an escalating amount of references. This can be considered one of the first –if not the very first- steps taken by Microsoft in order to materialize an approach towards ubiquitous computing. However, the perspective of having a proprietary solution as a means to solve, or at least try to solve, the challenges related to an environment such as the one presented here can be deemed as rather unappealing, as the amount and heterogeneity of the devices implied in a ubiquitous communication can expand beyond the speed the architecture can be adapted, and a proprietary solution would require to pay a royalty to a third company, which is not a good business for the other manufactures. This architecture was superseded by Microsoft´s .NET Remoting first and by Windows Communication Foundation after some more time [Microsoft 12].

2.2.4 Open issues and new approaches towards middleware.

The State of the Art in middleware ubiquitous architectures has an underlying weakness from where all the other can be considered to come: it is still at a very early stage of development, due to the fact of using many technologies and paradigms (Wireless Sensor Networks, Service Oriented Computing, Semantic Knowledge Management, motes...) that either are also in early stages of development or are mature enough but are not widely used yet. Several issues are pointed out in the following pages of this Final Degree Dissertation.

2.2.4.1 Lack of standardization and integration of middleware architectures.

The *status quo* in middleware ubiquitous, pervasive architectures (and, to a greater or a lesser extent, in most of the areas related with the new paradigms linked to these architectures) can be compared to the one that was prevalent in the computer packet switching industry in the late 60s or early 70s: a mosaic of proposals with very sparse utilization areas, where there is not a dominant solution above all the others. Wearable middleware architectures presented here have all made their contributions to the development of the others, or to the development of newer ones, but none of them have had a preponderant position in the market or for telecommunications and computer science researchers, and while some of them may have global consortiums of companies behind them (such is the case of UPnP), their original purpose or even their final objectives may not be directed towards the consecution of a middleware architecture properly conceived for paradigms as the Internet of Things, but just taking a part of it in a rather accidental way.



ubiquitous middleware architectures

You searched for: ubiquitous middleware architectures

443 Results returned

CONTENT TYPE

- Conference Publications (416)
- Journals & Magazines (26)
- Early Access Articles (1)

PUBLICATION YEAR

Single Year [✓ Range]

2000 2012

From: 2000

To: 2012

AUTHOR

Results per page 25 Sort by: Relevance

Select All on Page | Deselect All « First | 1 2 3 4 5 >> Last »

Multi-layered architecture of middleware for ubiquitous robot
In-Bae Jeong; Jong-Hwan Kim
Systems, Man and Cybernetics, 2008. SMC 2008. IEEE International Conference on
Digital Object Identifier: 10.1109/ICSMC.2008.4811836
Publication Year: 2008 , Page(s): 3479 - 3484
IEEE CONFERENCE PUBLICATIONS
Quick Abstract | PDF (481 KB)

Ubiquitous services for senior citizens service architecture and middleware

Figure 16: a IEEEXplore search for "ubiquitous middleware architectures" returns 443 results.

This can be explained as the evolution of the different services and products that had been marketed by the companies or research institutions implied in all these works; often, the creation of a middleware for a ubiquitous environment was just a small step forward in their former communication solutions –that were still carrying most of the particularities coded before, rather than having to start a new development from scratch-. This model may have been useful in the adoption of new paradigmatic ideas with little effort, thus guaranteeing a smooth transaction from a distributed middleware to a ubiquitous one, but it is nothing but a burden when the development demands to be pushed to its limits in order to have a fully pervasive, ubiquitous middleware, as many early traits will be kept and, what is worse, with their own particular and abundant differences: though ubiquitous middleware architectures tend to share some entities (a management hub, hardware-abstraction tools, peripheral nodes) their inner architectures and hierarchies are unrelated one to the other, making any effort to integrate all of them a titanic task. And even if they are integrated, there could be concerning consequences about composing a system with subsystems not defined to work together that should be measured -for example, having a database with user information and a Wireless Sensor Network to transmit data-; implementations that are able to work seamlessly without conflict must be integrated [Greenfield 06].

As far as it can be seen, this problem can be tackled in two different directions: either a centralised initiative is carried out by institutions or consortiums above all the smaller companies, but with their support and commitment, to define an ultimate model for a middleware suitable enough for the new paradigms –which could solve once and for all middleware integration problems, but with the cost of having to modify all the already existing ubiquitous middleware architectures to conform the new standard-, or a particular integration solution is implemented any time several different middleware architectures are put to work together –and therefore an integration effort has to be done every time that a compromise solution has to be obtained, albeit on a much smaller scale and just for a particular solution which will require re-thinking any other time a similar challenge is tackled-.

2.2.4.2 Lack of focus on security and privacy.

There does not seem to be any concern in making the presented middleware architectures at least as secure and privacy-concerned as their non-ubiquitous partners are. However, since the network traffic that can be generated is potentially much larger under ubiquitous conditions than in more conventional scenarios (it has to be taken into account, for example, a smart city with thousands of sensors scattered throughout all its buildings or green areas; such a project is already a tangible reality and can be easily exported, providing that enough interest and funding are available for local administrations, as for the SmartSantader case [SmartSantande 12]) not only because of the infrastructure used for obtaining information but also for its processing and share, huge steps towards enforcing more strict security measures should be taken. Unfortunately, none of these middleware architectures makes any effort in becoming security-aware, let alone implementing any effective treatment to counter any possible security thread. It can be argued that legacy solutions were too busy struggling to have a kind of architecture ubiquitous enough in the first place, so developing any security effort could have been an unbearable challenge for that time; however, the general consensus around this topic, even years before the arrival of Wireless Sensor Networks –that are clearly spearheading the efforts in ubiquitous computing- is that any network or piece of software that is not secure is effectively unusable, for it creates a vacuum of insecurity where disadvantages and risks of using networking and middleware technology overweight the advantages and benefits that they may offer.

Furthermore, the middleware architectures presented before find its natural field of usage within technologic environments, where data obtained will usually be network directions, device descriptions or indoor readings. The idea is that the collected information usually poses no interest for a hacker –or in fact, any third party not being part of the ubiquitous network- because it is bound to be used as part of a low-level utility. However, as it has already seen before, and as it will be exposed in chapter 4, ubiquitous middleware architectures could have at their ends sensors and devices that are actually reading critical and very private information about one person, from an identity card number to their body temperature; it becomes crystal clear that such data must be preserved from any external or unwanted agent.

Finally, the nature of the sort of networks that ubiquitous middleware may be embedded in makes them more vulnerable to attacks: ideally, a pervasive and ubiquitous system could require that it must remain operational 24 hours per week, 365 days per year, thus exposing critical information under a much wider time window that many conventional networks will do. That is not any imaginary device; right now there are many research projects with appliances with such applications born in mind when designed.

Figure 17: ubiquitous device from the University of Lübeck [Lübeck 12].

In order to solve this problem, a two-way solution must be conceived. First of all, security must be greatly improved by the side of ubiquitous computing in general and middleware in particular. Several steps are being taken in order to accomplish this: nSOM middleware architecture incorporates a security subcomponent which proves that adding security on ubiquitous middleware is possible and desirable. Plus, some other initiatives are taking place to secure sensitive data in the field of ubiquitous health care [Rosslin&11].

Secondly, final users of ubiquitous technology are expected to be people who do not necessarily have a good grasp of any sort of technology. Therefore, in case that the ubiquitous systems needs user intervention any time (of course, the less times the better, since a ubiquitous and pervasive system should require as little attention from the final user as possible), the user should have been educated before in its own usage as it is done under conventional networking environments, not revealing more information that the strictly required and being conscious of the possible presence of malware in their everyday ubiquitous products.

2.2.4.3 Struggle to adapt to present and future component heterogeneity.

One of the key functionalities of middleware is abstracting any higher layer from the heterogeneity and complexity of the underlying components of the architecture. It is a feature that is taken for granted in a regular networking environment, but, although it is usually done so under ubiquitous conditions, sometimes it feels as if some of the devices used in the systems need to be taken into account when the capabilities or the architecture are quoted. When this issue happens, it cannot be said that middleware is abstracting the heterogeneity of the hardware below the application layer, as it has to be considered when an application is developed. It is so because there is a certain degree of ambiguity about how to define and what to include as an element properly belonging to this paradigm: what makes a gadget or any other hardware appliance wearable or not (for example, while a mobile phone is universally regarded as a wearable device, a netbook is not granted the same status, despite being designed for being portable), what can be considered ubiquitous or not (a mote is clearly a device design with ubiquitous applications in mind, but then again, a mobile phone may not be considered as such, because its energy requirements, capabilities and functionalities surpass the ones expected on a device involved in a pervasive, ubiquitous system).

In spite of having proposed exact definitions of these concepts, in many cases they are either too narrow or become rapidly obsolete –one of the most widely accepted, Professor Steve Mann´s definition of what a wearable computer is [Mann 98], "*a computer that is subsumed into the personal space of the user, controlled by the user, and has both operational and interactional constancy, i.e. is always on and always accessible*" was done back in 1998 and can be challenged today under a ubiquitous and pervasive perspective, or can be considered just too imprecise-. An updated, accurate definition of what is wearable, ubiquitous and/or pervasive would come in handy under these conditions, and if it is done by an international standardization organism or consortium, it will become a good start point to define terms accurately.

Besides, subsystems involved in a so-called ubiquitous system must keep its functional and non-functional requirements on a minimum level. In order to achieve this, a policy could be followed where the performance of the middleware is dependent on the least electronically capable of all the devices involved in the architecture; that is, having as the performance and functional and non-functional requirements the ones determined by the bottleneck of the system.

2.2.4.4 New approaches towards middleware.

Thanks to the research efforts put during last years, and to the lessons learned from the legacy solutions in middleware as well, today there are current new approaches to the sort of middleware that can be fully exploited in the sort of environment this Final Degree Project is interested in. Although these tendencies are mostly theoretical schemes, unwilling to name the elements or the hierarchy of modern ubiquitous middleware architecture, they have several critical points in common, and are put into practice whenever a new middleware is designed:

- ***De facto*, not just *de jure*, abstraction from hardware-related layers**. In the former middleware architectures, even if hardware devices do not have to be taken into account when describing their architectures, their capabilities and restrictions are present in the expositions of their performance or their general layout (pervasive devices or hardware energy depletion is mentioned, for example). However, new trends abstract themselves for any consideration:

middleware architectures will just consider components and entities, where it is usually difficult to recognize when an element is a piece of hardware or software, as it was done before, albeit in a shy way, with middleware architectures that tried to be service-centric. Thus, it is simpler to ensure the portability of the ubiquitous middleware from one placement to another, and it even may be possible to have different pieces of equipment assuming different roles each time, or interchanging roles in case one the components of the ubiquitous middleware architecture has a power failure or is disabled for any other reason (security loopholes, etc.).

- **Bolder and wider view of ubiquitous (and middleware) architectures**. Rather than having an architecture for a very specific purpose (enabling ubiquitous printer capabilities, energy management) in a tiny environment (a room, an office), new approaches tend to think of ubiquitous middleware as something almost unlimited in its deployment and its extension; their position is that if the conventional Internet started with only several nodes and it expanded as the Internet as we know it today, ubiquitous architectures (and being part of them, ubiquitous middleware) should dare to suggest solutions and functionalities than can be applied to a varying amount of scenarios: outdoor surveillance, indoor sports, facility monitoring, etc. In addition to it, architecture must prove to have a share of features that can be regarded as unusual or reserved for conventional networking: Quality of Service measurement, scalability, reliability and robustness, security, etc. For its performance must be, if not as fast in data delivery as a conventional cabled or wireless network architecture, at least offering an equivalent degree of guarantees.

- **Enriched information as an improvement and a necessity**. New functionalities regarding the treatment of information (Semantic Web, Web Ontology Language) are open for any kind of environment; however, ubiquitous, pervasive middleware architectures will benefit the most, because the more efficient, the more data-refined and the more accuracy they are capable of showing when handling a client request, the less further requesting, resource consumption and wireless sensor network information interchange will be needed, resulting in a better overall performance and a higher degree of availability of the ubiquitous middleware resources.

In order to study the more recent approaches towards ubiquitous middleware, it has to be taken into account that they will often overlap with the idea of programming models for Wireless Sensor Networks.

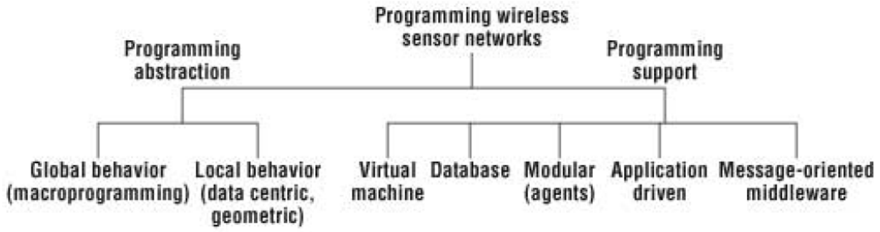

Programming wireless sensor networks
Programming abstraction
Programming support
Global behavior (macroprogramming)
Local behavior (data centric, geometric)
Virtual machine
Database
Modular (agents)
Application driven
Message-oriented middleware

Figure 18: taxonomy of programming models [Hadim&06].

As for programming Wireless Sensor Networks, there are two different programming points of view that can be observed in the previous taxonomy: one is **programming support**, named like that because it manages the provision of systems, services and execution time mechanisms, along with the reliable code distribution, safe code execution and other application-specific services. The other point of view would be **programming abstraction**, which is related with the way the Wireless Sensor Network is perceived externally, introducing concepts and notions about sensor data and the nodes belonging to the network.

Along with these major categories when programming Wireless Sensor Networks, there are some others that can be included within the former wider two [Vicente 10], [Familiar 09], more particular in their approach. The five ones encased in the category of programming support are the following:

1. **Virtual machine**: this point of view is composed by Virtual Machines, mobile agents and code interpreters. Compared with the other points of view, they stand out for their flexibility, allowing the programmers and developers the implementation of applications divided into smaller modules that are able to be distributed throughout the network by the system. The paths of distribution will be taken by previously designed algorithms, bent on reducing energy and resource depletion.

   There are many modern implementations that can be displayed as examples. For example, the **Maté** initiative is been operational since 2002 in an effort to provide a bytecodes interpreter that runs on TinyOS as the operating system. Written in the already mentioned nesC, it makes use of a model that will trigger an execution of an action after an event has been received. In addition to that, capsules downloaded to the network nodes with numbered versions make it possible to better update the code. The **Squawk** Virtual Machine can be exemplified as another virtual machine-based solution; however, it has been designed keeping in mind mostly Sun SPOT motes.

2. **Database**: this point of view regards the network as a virtual database. In this view, a very simple interface is offered for information exchange, using queries to extract information from the sensors placed in the Wireless Sensor Networks. However, it has an important drawback: this point of view does not offer any support for real-time applications, providing only estimated data, making impossible the inferring of relations between space and time between events.

   An implementation of this kind of programming support is **Cougar**, for the whole Wireless Sensor Network is just a relational database under this programming model. Operation management of the wireless Sensor Network is carried out with a query language similar to SQL. Cougar is able to characterize any node of the network as a database generating structured registries with several fields. Abstract data types are used to model signal processing functions, and an energy saving system based on request distribution among the nodes will be employed as well. Another case is **TinyDB**, developed by Madden et al. in 2010, which makes use of a queries processing system in order to get the data from the Wireless

Sensor Network. It is not a coincidence that it shares part of its name with TinyOS, as this is the operating system this programming model is mounted upon. TinyDB keeps a virtual database with information about the kind of sensors that is put into a use, node identification and the battery charge that still remains for each of them. Furthermore, TinyDB will employ an interface similar to SQL´s so as to better extract the required data. However TinyDB is limited by its mandatory use of TinyOS; no other operating systems are allowed, and the code uploaded to the motes must be written in C as the programming language.

3. **Modular programming (agents)**. This approach uses the mobile nature of the code under its principles to its advantage. As it happened with the Virtual Machine approach, the modularity of the applications makes it easier to inject and distribute mobile code, and propagating small modules throughout the network make them less energy-demanding.

   As examples of this programming model, Impala and the Smart Messages Project can be put forward. **Impala** was first shown by Liu et al. in 2003 as an architectonic model that offered mobility, transparency and quality changing from one protocol to another and adapting applications in real-time, thus preventing many node failures and runtime errors. However, Impala was only available for Linux-based IPAQ and Pocket PC devices from Hewlett-Packard and Compaq, making its used too narrow in this context. **Smart Messages Project**, on the other hand, was developed by Kang and other researchers circa 2004 and puts forward a distributed model called Cooperative Computing, where migratory execution units called smart messages are defined to cooperate for a common goal.

4. **Application driven**. Here, unlike the other points of view, an architecture following a stack model is offered. This is advantageous for the developers, as it offers them the chance to tune up the application to a level unthinkable with other solutions. Since the network management is done by the very applications, Quality of service or QoS can be improved, according to the needs of the application.

   An application driven programming models is **MiLAN** (Middleware Linking Applications and Networks), developed by Murphy and Heinzelman in 2002 [García-Hernando&08]. MiLAN focuses on high-level details, using a characterized interface, and allowing network applications to specify their particular needs of Quality of Service (QoS) and trim network features in order to optimize its performance without ignoring the needs of them. Commonly, MiLAN will select the group of nodes that are compliant to the specifications of the QoS required by the application running at that moment by means of network plug-ins that will determine the group of nodes that better accomplish the requested duties. Under MiLAN, networks can be configured in a very accurate way, due to the fact that the group of nodes is chosen by making use of its extended architecture (comprising the network protocol stack, the abstraction layer and the network plug-ins) and specialized graphics responsible for adding changes based on the needs of the applications.

5. **Message-Oriented Middleware (MOM)**. This is the model that probably best suits programming under Wireless Sensor Networks conditions. A mechanism of publication and subscription to services (*publish-subscribe* model) is put into practice in order to facilitate communications between nodes and base stations. Plus, this model enables asynchronous communication, allowing for a flexible model of communications between information creator and receiver.

   Two representative implementations of this kind of model are Mires and SensorBus. **Mires** was developed by Souto and other researchers in 2004; it makes use of activation messages to put to a use a communication infrastructure based on a component of publication and subscription (or more accurately, a publish subscriber service). This component will synchronize communications among middleware services and will make the system work properly. In addition to this, a routing component and data aggregation service are offered; this latter service let the user point out both how the data is going to be added, along with the relation between harvested and added data. As it happened in TinyDB case, Mires makes use of TinyOS as the operating system and C as the programming language. **SensorBus** was designed in 2005 by Ribeiro et al. It also implements the publication-subscription paradigm (this time, this mechanism is anonymous, asynchronous and multicast). SensorBus makes use of a key component called events producer that will publish the kind of events that are going to be available for the event-consuming components of the system. The event-consuming component will just subscribe to it and will receive notifications from the event generators about that kind of subjects that will be taking place.

On the other hand, other categories falling under the orbit of programming abstraction are:

1. **Global behavior or macroprogramming**. This point of view tries to consider the global behavior of the whole Wireless Sensor Network. Instead of programming having only individual nodes in mind, the wireless Sensor Network is programmed as whole, black-boxed subsystem, with its expected behavior as a whole according to a high-level specification that generates automatically the code that enables the behavior of every node, thus freeing application development from treating the low architectural levels of the nodes that compose a network.

   Two examples can be put forward belonging to the macroprogramming perspective: Kairos and Semantic Streams. **Kairos** was developed by Gummadi et al. in 2005; it makes use of the concept of “divide and conquer”: the global behaviour of the whole network will be broken into entities called subprograms, and these latter can be executed locally in every node. Since this process will be carried out not only in compilation time, but also in running time, the developer is left charged just with handling a few primitives. According to the expectable middleware functionalities, Kairos manages to insulate higher layers from lower level details. Besides, Kairos provides three important abstractions for the programming language that can be regarded as tools: a) Manipulation of the behaviour of the network nodes by abstracting them individually, b) An updated list of the neighbouring nodes of a particular node, by defining a collection of all the other nodes at one radio-jump of distance and c) The reading of variables from specific nodes thanks to a data access mechanism. Finally, Kairos makes possible the selection of the way the processes of each node are synchronized: either in a more flexible manner (loose synchronization) or in a strictly defined one (tight synchronization). It is up to the programmer to decide how to use these processes to have an efficient network without any system overload.

Semantic Streams, on the other hand, proves to have a little more intelligent than the other programming perspectives: it enables the user to make declarative requests about semantic interpretations of the information gathered by the sensors, where data has a meaning inside a context.

2. **Local behavior (or geometric, data-centric)**. This point of view takes the opposite path: nodes are taken into account individually, inside the distributed network. This local behavior is focused on two features: firstly, the nature of the data info obtained by the sensors; secondly, the specific location of the network element. Any request for a data reading (moisture, temperature, luminosity) of a particular spot inside the environment could be an example of this category.

   Another two examples following local behavior programming paradigm can be offered here: Abstract Regions and EnviroTalk. **Abstract Regions** was conceived by Welsh and other researchers in 2004; it is nothing more than a group of communication, general-purpose service primitives that provides data aggregation, addressing, data sharing and data reduction in the local regions of the network. Other of the main ideas behind Abstract Regions is shifting the focus on local computation to the detriment of radio communication, thus reducing the bandwidth used in communications (at the expense of requiring more processing activity). Data processing and aggregation are done at the local level by a group of cooperative nodes that communicate among them. As far as a hierarchy is concerned, data are sent from the nodes to the base station without any go-between, in another effort to reduce bandwidth depletion. Any object that requires a very precise tracking, which will aggregate readings from all the nodes nearby, is an example of how a system could benefit from this implementation. **EnviroTalk**, on the other hand, was designed by Abdelzaher et al. in 2004. It tries to create a data-centric inspired naming system called “attribute-based naming” that makes use of context labels. This paradigm is original in the fact that addressing and routing are not based on the destination node, but on the content of the requested data. The dynamic behavior of a mobile object and the nodes that belong to the network as well can be accurately followed, making this system fairly suitable for tracking or environmental monitoring solutions. However, as it happened in other examples of programming paradigms, enviroTalk makes use of TinyOS as the operating system, and therefore it is limited in the usage of programs.

Finally, all new perspectives and contributions that were mentioned before, along with the way they are integrated in the environments of interest, will be reviewed in depth in the following parts of this Final Degree Dissertation.

2.3 Novel tendencies in ubiquitous computing.

The main difference of these tendencies with “conventional” or “regular” computing is that they offer a completely new perspective to build up middleware architectures, without using almost any of the legacy ideas (and therefore, without almost any use of legacy middleware systems) introduced before. Also, they are focused on providing some extra intelligence to the systems and the middleware layer: data is no longer a stream of chunks of something alien to the network; on the contrary, it will be accessed and consulted by the network itself and the semantics of the content it will make a difference for its treatment.

It is also worth mentioning that component-centric computing is on the decline; more abstract paradigms that are centred on interchangeable software components and services are gaining momentum in the field of ubiquitous computing.

2.3.1 Pervasive Computing and the Internet of Things.

These terms, along with ubiquitous computing and the Internet of the Future, are often used interchangeably. They refer to a final goal and a sort of idea of how systems are supposed to be in the mid or long term, rather than a solid, well-defined paradigm. Commonly, pervasive computing and the Internet of Things are referred to the same core concept: providing an environment of a variable -but usually wide- size with nodes, sensors, and electronically augmented devices so as a virtual map of the physical world is obtained, learning not only the location of an object (or “thing”), but also their most prominent features (temperature, luminosity, battery levels, etc.) and in general terms, getting an idea of everything that can be measured from an element at a electronic level (relatively unchanging data as size, weight and height of the object the sensor is attached to, the kind of garment that the object has been manufactured with, etc. are data that can be hardcoded in the device as repeated readings or events are unlikely to give varying lectures).

Again, when talking about pervasive computing and the Internet of Things, it is mandatory to mention Mark Weiser (1952-1999). In a 1996 presentation, Weiser predicted that computing would become increasingly ubiquitous, with an inexorable shift of focus from a computer-centric scenario (mainframes used by many people as a way to interact with computers), to a human-computer equality one (Personal Computers, laptops) and finally to arrive to a human-centric scenario where there would be many computers used by each person, although not necessarily in an attention-demanding way.

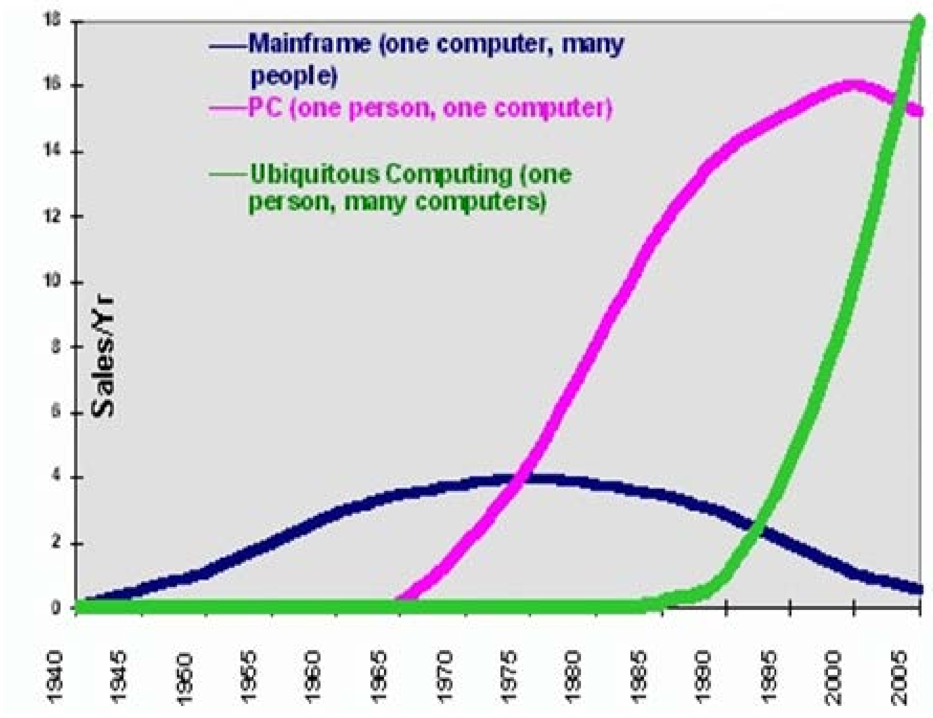


Figure 19: foreseen computing trends by Mark Weiser [ISGTW 07].

This conclusion has been acknowledged by other people as well, some of them not even belonging to the field of technology but sociology: Manuel Castells argues in his book *The Rise of the Network Society* that the shift from one paradigm to another is already taking place, from already-decentralised computers towards entirely pervasive computing, as the Internet best exemplifies [Castells 96].

The Internet of Things, of the other hand, has received many overlapping definitions that go beyond the idea of having a piece of environment fully augmented with miniaturized electronics; for example, it has been labelled by the CASAGRAS (Coordination and support action for global RFID-related activities and standardisation) project, part of the CORDIS (Community Research and Development Information Service, an informational portal supportive of European research and innovation cooperation) initiative as "*A global network infrastructure, linking physical and virtual objects through the exploitation of data capture and communication capabilities. This infrastructure includes existing and evolving Internet and network developments. It will offer specific object-identification, sensor and connection capability as the basis for the development of independent cooperative services and applications. These will be characterised by a high degree of autonomous data capture, event transfer, network connectivity and interoperability*" [CORDIS 08], and a company as important as SAP has described it as "*A world where physical objects are seamlessly integrated into the information network, and where the physical objects can become active participants in business processes. Services are available to interact with these 'smart objects' over the Internet, query and change their state and any information associated with them, taking into account security and privacy issues*" [Haller 09]. At the hardware level, there is an important degree of heterogeneity because of the already mentioned ambiguity over the definition of what is "wearable" or "ubiquitous". Wireless Sensor Networks, with all their radio protocols, motes and architectures are well-represented, but there are also other technologies that are considered to forerun the efforts of spreading the Internet of Things, notably RFID (Radio Frequency IDentification). RFID has often paved the way for other visions related with Internet of Things. It can be defined as a generic term used to describe a system that transmits the identity of an object or a person wirelessly, using radio waves [Aimglobal 12]. An RFID system will usually be composed by a RFID tag or transponder containing all the relevant data from the device the tag is attached to, an antenna or an interrogating device that will either request data or wait for an event (for example, a RFID tag passing nearby) to be triggered, and, more often than not, an infrastructure composed by a computer and software that will treat the data received if the interrogating device asks for it or receives as an event.

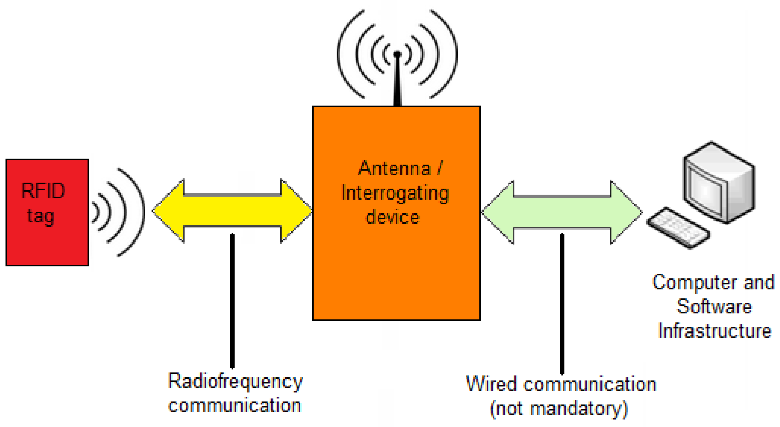

RFID tag
Antenna / Interrogating device
Computer and Software Infrastructure
Radiofrequency communication
Wired communication (not mandatory)

Figure 20: a RFID infrastructure.

All in all, the Internet of Things is expected to expand rapidly in the next years, thanks to the possibility of successfully solving the challenges that new services demand to be solved at a reasonable pace in time, as it can be seen on the graph located below:

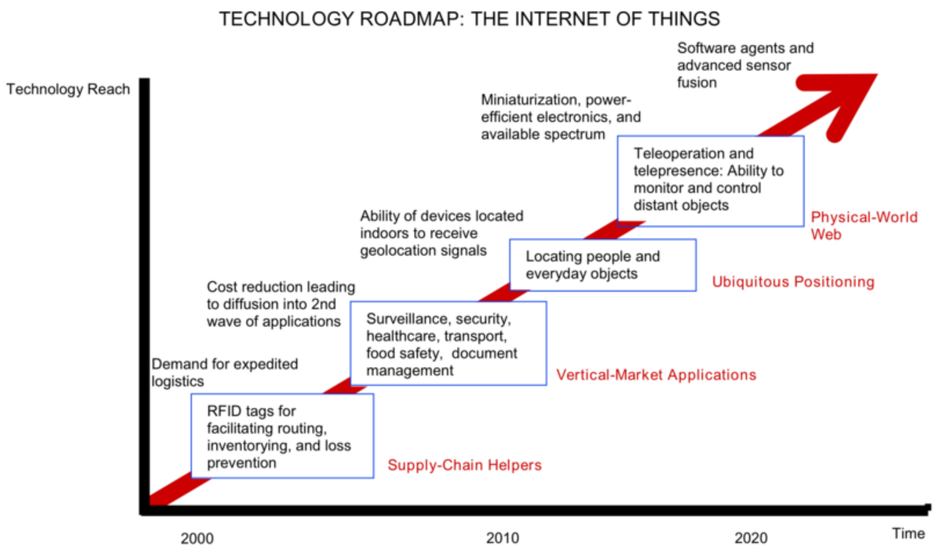


Figure 21: roadmap for the Internet of Things, from [SRI 08].

2.3.2 Service Oriented Computing and Service Oriented Architecture.

Service Oriented Computing or SOC is another novel paradigm in the world of middleware that has a particularly strong saying over matters in the most recent proposals of middleware for Wireless Sensor Networks. It is based on Service Oriented Architecture, a paradigm also extended to non-ubiquitous scenarios, based on the publication and discovery of interfaces that can be used to communicate devices with each other.

The main focus of Service Oriented Computing is in using very basic services in order to develop and implement the most efficient possible distributed applications (meaning by that being simple, fast and low-cost) under heterogeneous environments [York 03]. These very basic services are not dependent on any hardware condition or any other purpose; they are autonomous entities that may –or may not, depending on the requirements of the network or the user at the moment any service is requested- be discovered and published by the environment where they are taking place, and once this happens they usually will interchange data by means of request/response mechanisms.

What is more, code already present from the application may be reused by these services so as to compose brand new services, independent on their own. This offers a chance to develop new services and, by proxy, new applications created upon the new services discovered, instead of having to start the work on them from scratch.

A huge improvement over many other paradigms is that Service Oriented Computing does not use any particular programming language or operating system; SOC usually uses protocols and standardized languages based on XML, and implements interfaces capable of abstracting programmers from lower layers. Therefore, developments can be undertaken equally under by using C, Java, TinyOS or Contiki. This independence makes its way for internetworking protocols, regardless if they are related to the Internet or the TCP/IP stack or are deeply dissimilar in all their parts (UMTS, xDSL, Bluetooth).

One kind of the most developed services under the paradigm of Service Oriented Computing is the idea of a Web Service. Web Services are used for development and execution of network-distributed negotiation processes, and are accessed in a standardized way, with the usage of regulated interfaces and protocols. Their resources to get the information served to the final user are vast:

- Web Services can use IP with other transport protocol as their mean of communication, along with open standards such as Simple Object Access Protocol or SOAP (although it is being increasingly pushed back by Web Services based on REST -REpresentational State Transfer-).

- Standard service description language as WSDL (Web Services Description Language) can be used for service publication and definition, along with BPEL (Business Process Execution Language) for service orchestration.

In order to better acknowledge the Service Oriented Computing paradigm, a model of the Service Oriented Architecture is provided below. According to the view of Mike P. Papazoglou and van den Heuvel, who exposed a joint model in 2007, there are three main members in SOA architecture: service provider, service discovery agencies and a client or service requester. These three members will engage in three operations: publishing, finding and binding. The interaction will occur between two software agents called service requesters or clients (software agents that must be able to find the description of the requested service and bind it) and service providers (software agents responsible for publishing a service description when clients request for service distribution), being able to behave simultaneously as clients and providers, and it will be done with message interchanges.

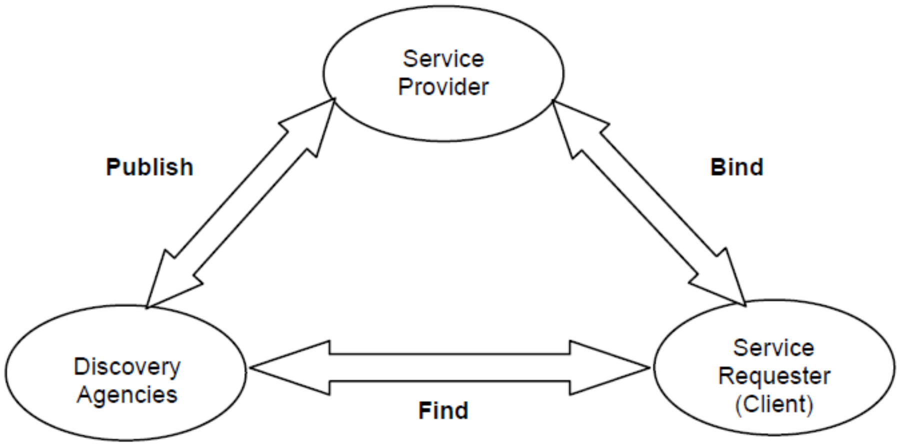


Figure 22: a representation for a basic Service Oriented Architecture.

Following the most notorious features of SOC and SOA paradigms several middleware architectures have been developed, focused mostly on their implementation on Wireless Sensor Networks. Because of the key role that WSNs usually play in systems based on the Internet of Things, two them are being described.

2.3.2.1 Persistent queries-oriented SOC.

The main goal of this middleware implementation is minimizing the number of persistent queries done during the service composition process in order to save energy. Thus, a selection of composed services that can guarantee the least possible energetic depletion must be ensured.

Under this perspective, a routing protocol is put forward for query transmission under a Service Oriented Architecture. This routing protocol must define not only a certain path throughout the Wireless Sensor Network from a regular mote to the sink/base station (usually, WSNs have just one mote as the base station) but the route must also include the service providers. In addition to this, persistent queries-oriented SOC proposes two algorithms designed to optimise the Wireless Sensor Network performance:

- An algorithm used in order to trim to its minimum the number of service requests in a service composition, a feature that should come in handy to reduce the energy consumption in the nodes of the network.
- A dynamic programming algorithm deployed with the purpose of reducing the whole cost of the service composition (in terms of energy, time, in a holistic view of the needed resources), with the expectable drop in query transmission time.

In this kind of development, the deployed architecture has three different levels: a service query level, a proper service level and a service composition level. When the system has to perform any required action, once the required services have been identified, the routing protocol will find a route between the nodes capable of providing the services and the nodes that requested them, in addition to a path between two adjacent service providers. During the time used for the request a new necessity could spring up: the service composition procedure could need a change if any essential node has reverted to an idle state or is down. Should this happen, the query will be taken again once an alternative node capable of providing the service has been located.

2.3.2.2 Oasis.

As middleware architectures do, Oasis is a Service Oriented Architecture designed for Wireless Sensor Networks that makes an effort in insulating all the technical heterogeneity that can be expected from a Wireless Sensor Network. Although locally synchronous, Oasis makes use of a general asynchronous framework complemented with a user-friendly environment that allows the service composition to be treated as if services were complete applications [Hadim&06]. Any activity present on a WSN, along with the behaviour of its components, will be treated as an independent service, and the services will be offered by accurately defined interfaces.

The offered services by the middleware services package offer and wide range of operations supporting the inner functionalities of the Wireless Sensor Network that receives them: internal operations between nodes, discovery and communication of services and maintenance of them. On the other hand, Oasis´ framework can be used to develop data stream applications which, given the fact that Oasis will be usually mounted upon a ubiquitous environment system, can be programmed with many purposes: vehicle location, fire detection, etc.

2.3.3 Semantic Knowledge Management and the Semantic Web.

Semantic Knowledge Management is another new paradigm used both for ubiquitous and non-ubiquitous environments. Usually, the content that is sent through a Wireless Sensor Network (and a more conventional network as well) has a very little chance to be interpreted, there is no “intelligence” to understand its meaning or learn from it. On the contrary, under a Semantic Knowledge system the information from the network can be managed more efficiently, and the data extracted is not “raw” information, but “refined” information that has a meaning by itself. Service Knowledge Management can also be defined as a collection of practices that seeks to classify the transiting information by taking account of the knowledge contained in the pieces of information, transforming it (using standardized technologies in the process) and delivering it to the users that requested it in the first place. This content will be classified according to a format, which will usually be related to the XML suite [Davies&08].

Semantic Knowledge Management is critical for the construction and maturation of the Semantic Web. This term was first coined by the father of the World Wide Web, Sir Tim Berners-Lee, in 2001; he defined the Semantic Web as “*a web of data that can be processed directly and indirectly by machines*”. The idea supporting its development is that until now, despite their mobile or ubiquitous nature, applications won´t share or publish the acquired data, and will even discard it if a composed service that requires several pieces of information has one or some of them corrupted or unusable (according to W3C, the World Wide Web Consortium, “*For example, I can see my bank statements on the web, and my photographs, and I can see my appointments in a calendar. But can I see my photos in a calendar to see what I was doing when I took them? Can I see bank statement lines in a calendar? Why not? Because we don't have a web of data. Because data is controlled by applications, and each application keeps it to itself*” [W3C 12]). With the Semantic Web, data will be shared among applications under a common framework, not unlike a ubiquitous Wireless Sensor Network, that will surpass any boundary artificially created by applications, enterprises or communities.

Like other paradigms, the Semantic Web will manage the semantic content with a Stack model that looks like this:

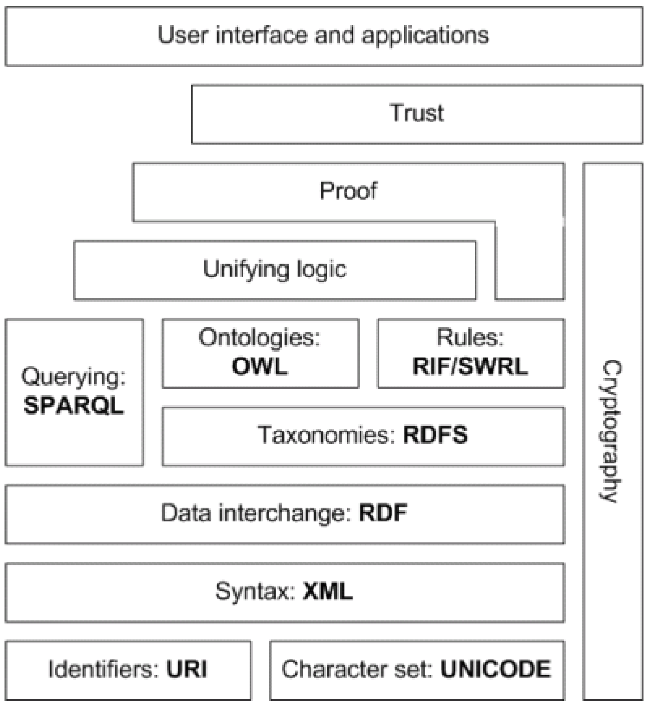


Figure 23: Semantic Web stack, as it appears on [Obitko 12].

The components of this model are, from the bottom to the top layer:

- **URI** (Uniform Resource Identifier) as the sort of identifier of a resource. This is a stream of characters that is giving a description of the object (usually a web page, but it could be another resource in a ubiquitous, pervasive environment), including both the location of the resource (URL or Uniform Resource Locator) and its name (URN, Uniform Resource Name). Characters used to display the information will be in **Unicode**, a format aiming to support all the different writing systems of the world.

- A syntax written in **XML** (eXtensible Markup Language). This is a meta-language used for description of the information regarding documents. Its wide use and its standardization purposes make it a good choice for this architecture

- A single layer for data interchange that makes use of **RDF** (Resource Description Framework), a W3C specification originally meant to be a metadata data model but used as a general method for conceptual description and modelling of information. RDF uses subject-predicate-object expressions (also known as triples) for its modelling of contextual approaches, where the subject is denoting the resource and the predicate displays features from the resource and the relationship between the resource and the object [Sikos 11].

- Taxonomies based on schemas of RDF or **RDFS**. RDFS is the vocabulary description language for RDF, a semantic extension of RDF itself that provides several mechanisms for the description of a cluster of resources, and the relationships built around them [W3C RDF, 2012]. As it will be seen later, RDF Schemas, along with OWL, are able to provide languages for expressing ontologies in the Semantic Web.

- A query language that makes use of RDF facilities called **SPARQL** (or the recursive acronym for SPARQL Protocol and RDF Query Language). This language, which is a direct equivalent of a query database language (to the point that Sir Tim Berners-Lee, currently the director of the W3C, claimed that "*trying to use the Semantic Web without SPARQL is like trying to use a relational database without SQL*" [W3C 12f]) has been designed to generate queries that will hide the details of data management, in an effort to lower costs and increase the robustness of data integration on the Web.

- Another crucial component from the architecture is a language to generate ontologies. While applications within a Semantic Web context can be implemented without ontologies [W3C 12], some other may need an agreement on common terminologies or even complex reasoning procedures that will require their use [W3C 12b]. As it was defined before, an ontology is a group of concepts and relationships used to describe and represent a field of knowledge. In this context of Semantic Web, ontologies are useful for classifying the terms present in an application, fix relationships among them and define possible limitations in those relationships. Along with RDF Schemas, the language most common to define ontologies and their relationships is **OWL** (Web Ontology Language). The second version of OWL defines itself as an ontology language for the Semantic Web with formally defined meaning. According to W3C, OWL 2 ontologies provide individuals, properties, classes and data values that can be stored as Semantic Web documents. OWL 2 ontologies are closely related to RDF: they can be used along documents written in RDF language, and the very OWL 2 ontologies are exchanged as RDF documents [W3C 12c].

- The next component makes reference to the need to define some rules for the intercommunication of applications. It is expected to become standardized sooner than later, but right now there are two main recommendations by the W3C: RIF and SWRL. **RIF** (Rule Interchange Format) is an XML-based language for expressing rules that computers can understand and execute [W3C 12e]. As it happens with other recommendations from W3C (such as OWL), RIF features a plethora of dialects for the design of a rule language in a particular implementation. **SWRL** (Semantic Web Rule Language), on the other hand, differs from RIF in the fact that RIF was designed, at least at first, as an interchange format for exchanging rules between rule systems, whereas SWRL is a rule language that was designed as a syntactic extension of OWL and does not take into account such facts [W3C 12d].
- Other layers that are present but do not necessarily make use of any already developed technologies are a Unifying logic layer and a Proof layer. They are expected to add robustness and homogeneity to the whole architecture, but are not accurately implemented yet. Along with them, a Trust layer can be added just below the application layer as well, in order to make requests about the service offered by the application.
- A transversal Cryptography layer for the communications is offer throughout almost all of the architecture layers to guarantee a minimum level of security.

It is expected that in the next years all these new concepts will be incorporated in ubiquitous computing as they are very close to the idea of pervasive and seamless integration of all imaginable applications and entities of the Semantic Web.

2.3.4 Open issues.

The issues that can be mentioned here, either related to the Internet of Things or the semantic Knowledge Management are related primarily to two facts: these new approaches are still under a mainly theoretical perspective and their development into actual implementations is still limited. In addition to that, it looks like a difficult task to change the already developed middleware architectures under other paradigms to make them compliant with these new ideas, although their usage is less frequent that other wider architectures that are under deep processes of change (for example, update from IPv4 to IPv6), which will make it somewhat easier to shift the focus.

There is some criticism regarding the achievements made by the Semantic Web so far. It is stated that the hype and expectations of 2001, when the term was first used, have largely waned, and the evolution that was foretold is a target yet to be accomplished (in 2006, Sir Tim Berners-Lee was quoted saying "*This simple idea... remains largely unrealized*"). Perhaps due to its underdevelopment, there are still some components of the Semantic Web that are not completely standard (RIF and SWRL) and others that right now are not even fully realized (Unifying logic, Proof), which adds unwanted uncertainty in an architecture that is supposed to be used as a pattern for the building of applications and the retrieval of other resources in a Semantic web context. In addition to it, the many versions –or dialects- of several W3C recommendations, such as OWL or RIF, could jeopardize the process of standardization and mutual understanding of the different applications present in the Semantic Web.

Finally, there is some concern over the censorship and the privacy policies that could be enforced by entities unrelated with pure technologic facets of the Semantic Web (governments, corporations). In a web where text-analyzing techniques are easy to be implemented, a much tighter control of the data flowing through a network can be sought; plus, the use of geo-location metadata reduces the privacy of the final user requesting a service.

2.4 Hardware platforms.

Many different kinds of user platforms can be considered as part of a ubiquitous system; however, the features mentioned before as typical of pervasive and ubiquitous devices must be relevant in the building of the platform and in the way its functionalities are carried out. Even though this may refine the search for ubiquitous hardware platforms, it is still an extremely wide field. In this Final Degree Dissertation stress will be placed in the most developed and relevant hardware platforms: motes usually found as part of Wireless Sensor Networks, along with some other examples of wearable devices that can be found as part of a final user component.

2.4.1 Classes of hardware platforms and manufacturers.

To begin with, a survey has been done regarding the most relevant manufacturers of motes. Motes will be playing a very important role in this Final Degree Dissertation, as they have been combined into a Wireless Sensor Network from where information will be harvested and processed in order to obtain several different services, as it will be shown. Besides, Wireless Sensor Networks are the most developed and most mature subsystem of the new paradigms previously seen: they provide a pervasive source of data, usually service-oriented, where semantic knowledge and management can be added; in addition to that, their small size and low energy requirements (compared to some other counterparts, like smart phones or tablets) make them suitable for an implementation of a system following the Internet of Things principles.

2.4.1.1 Sentilla.

Sentilla (previously known as MoteIV) used to manufacture, among other variety of products, a mote model, but as of 2012 their hardware production has been discontinued an only offer software solutions focused on data and power management and performance [Sentilla 12].

Figure 24: Sentilla company logo [Sentilla 12].

Their last mote development, Sentilla Tmote Create, was in fact a mote with what Sentilla called “Sentilla Point Runtime”, a runtime featuring Java technology, so in the end as far as software is considered, they were not much dissimilar to Oracle Sun SPOT motes, which will be reviewed in depth later. However, before that last development, Sentilla manufactured a different piece of hardware that deserves to be looked upon as a legacy solution: Tmote Sky mote.

2.4.1.1.1 Tmote Sky

Developed initially in the University of California, Berkeley, Tmote Sky motes were marketed as wireless, ultra-low powered devices especially advisable for tasks like application monitoring, Wireless Sensor Networks or fast-application prototypes. Also, they used a standardized USB interface in order to improve and facilitate application codification and debugging, a solution that has been adopted by other vendors (Sun Microsystems, for example, before being purchased by Oracle had already added a mini-USB interface to their own Sun SPOT motes). Tmote Sky motes had an integrated antenna with a 125 meters long coverage area, and a flash memory with a built-in software image that can be replaced in case the mote was malfunctioning.

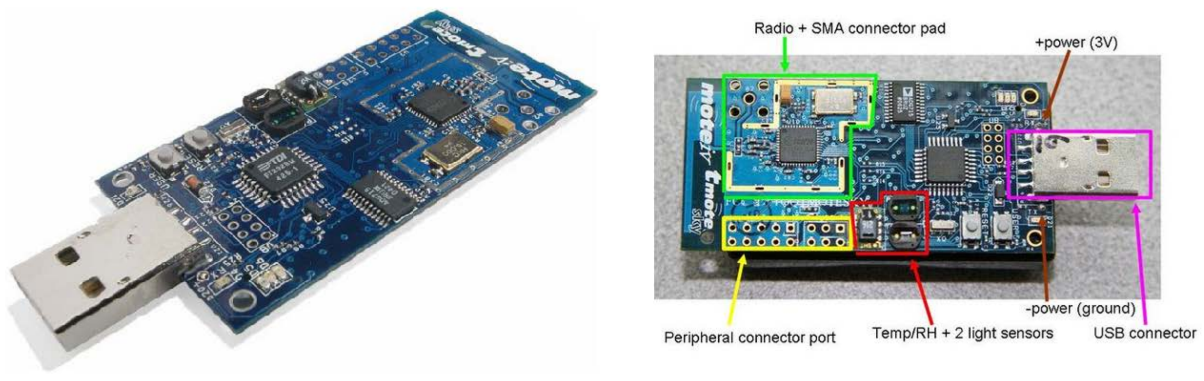

Radio + SMA connector pad
+power (3V)
-power (ground)
Peripheral connector port
Temp/RH + 2 light sensors
USB connector

Figure 25: Tmote Sky mote and components [González 10].

A chart with its most relevant technical features is presented following the previous figure:

Central Processing Unit		**Interfaces**	
Microcontroller	MSP430F1611	Digital	6 GPIO
RAM	10 KB RAM	Analogical	6 ADC
ROM	48 KB (Flash)	Serial	I^{2}C, SPI, USB
Speed	8 MHz	Sensors	Humidity, light and temperature
Operating System	TinyOS, Contiki-compatible.	Gateway Infrastructure	Ethernet
Radiofrequency		**Others**	
Transceiver	Chipcon CC2420	Battery	2 X AA Batteries
Band	2400 – 2483.5 MHz	Consumption (idle)	5.1 uA
Data speed	250 Kbps	Consumption (Rx/Tx)	19.5 mA / 21.8 mA
Modulation	O-QPSK	Price	As of 2008, 130 $ per unit (780 $ 10 units pack)
Range	125 m. (outdoor), 50 m. (indoor)	Gateway	Tmote Connect
Protocol	802.15.4	License	BSD Operating System
Security	Proper of 802.15.4	Extras	None

Chart 2: Tmote Sky technical features.

2.4.1.2 ETH-Zurich.

The Swiss Federal Institute of Technology, located in Zurich, has developed a mote design and implementation especially focused on the duties related with the Internet of Things and Pervasive Computing objectives [ETH 08b]. They are the result of the combined effort of the Computer Engineering and Networks Laboratory (TIK) and the Research Group for Distributed Systems. They are named BTNodes, and currently the third review of their hardware and software capabilities is being marketed, thus giving its name to the mote.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Figure 26: ETH logo [ETH 08b].

2.4.1.2.1 BTNode rev3

ETH claims that their BTNode motes are nothing but "*an autonomous wireless communication and computing platform based on a Bluetooth radio and a microcontroller*", stressing their value as demonstration platforms for research in mobile and ad-hoc connected networks (MANETs) and distributed, wireless sensor networks. Their internal architecture relays on the idea of using radio and Bluetooth interfaces in the same device.

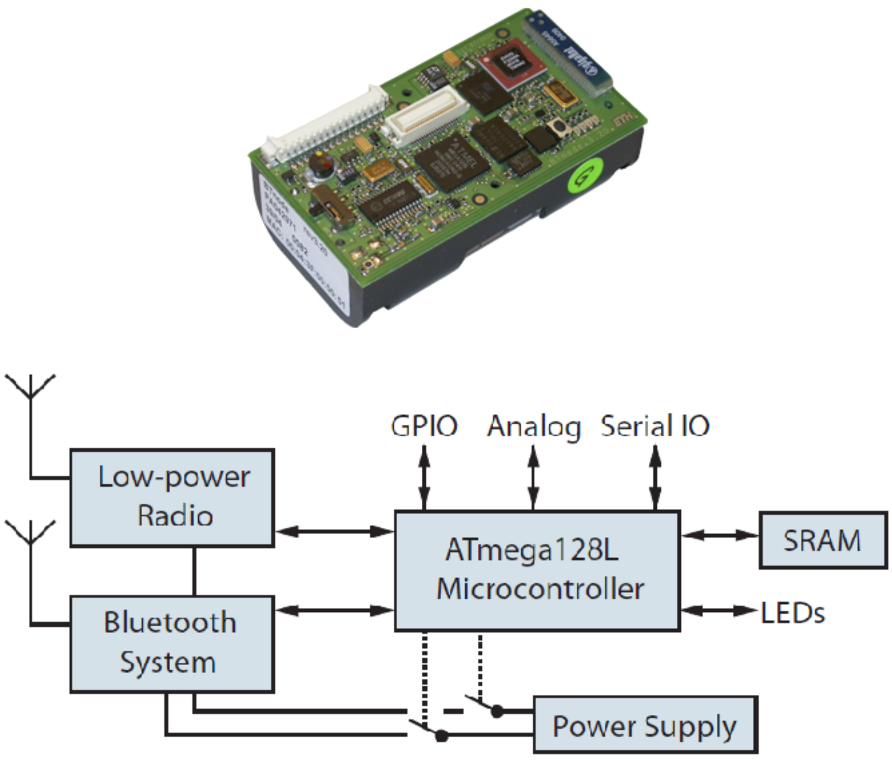


Figure 27: up, BTNode; down, BTNode parts [ETH 08].

It should be noted that both radio systems can be turned on and off separately, in order to reduce the power consumption of the device when it is under idle conditions. Alas, the low-power radio used here is the same that the one mounted in Mica2 Motes from Berkeley. The open source community is offered as a means of support.

BTnode rev3 most relevant characteristics are as follows:

Central Processing Unit		**Interfaces**	
Microcontroller	Atmel ATmega128L	Digital	GPIO
RAM	64+180 KB	Analogical	ADC
ROM	128 KB (Flash)	Serial	I²C, SPI, UART
Speed	8 MHz	Sensors	Humidity, visible light, infrared light, microphone, 2 axis accelerometer and temperature
Operating System	BTnut, TinyOS-compatible.	Gateway Infrastructure	None
Radiofrequency		**Others**	
Transceiver	Chipcon CC1000	Battery	2 X AA Batteries, external DC source (3,8-5V)
Band	ISM Band 433-915 MHz	Consumption (idle)	9.9 mA
Data speed	76.8 Kbps	Consumption (Rx/Tx)	105.6 mA
Modulation	FSK	Price	215 $ per unit
Range	+100 meters outdoor (with antenna attached)	Gateway	None
Protocol	Not defined	License	Propietary license
Security	Not defined	Extras	Radio Bluetooth 1.2 Zeevo BTSense plugin

Chart 3: BTNode technical features.

2.4.1.3 Libelium.

Libelium is a company originated in 2006 as a spin-off company from the University of Zaragoza, that currently is bent on products and services related with the Internet of Things; especially hardware ones. According to their web page, "*Libelium designs and manufactures hardware technology for the implementation of wireless sensor networks so that system integrators, engineering and consultancy companies can implement reliable Smart Cities solutions to end users within the minimum time to market*" [Libelium 12c].

libelium

wireless distributed communications

Figure 28: Libelium logo [Libelium 12].

Among their group of offered services and products, there is one mote design within the scope of this Final Degree Dissertation: the Waspmote.

2.4.1.3.1 Waspmote.

As the products of their competence, Waspmotes are been developed looking for the varied scenarios where the Internet of the Future is likely to have a say, as part of their Top 50 Sensor Applications for a Smarter World [Libelium 12]: applications in smart cities (noise urban maps, structural health), water treatment (water quality, water leakages), industrial control (indoor air quality, temperature monitoring) Among other achievements, Libelium claims that Waspmotes can work without being recharged for as long as one year, providing that the energy-saving Hibernate mode is enabled.

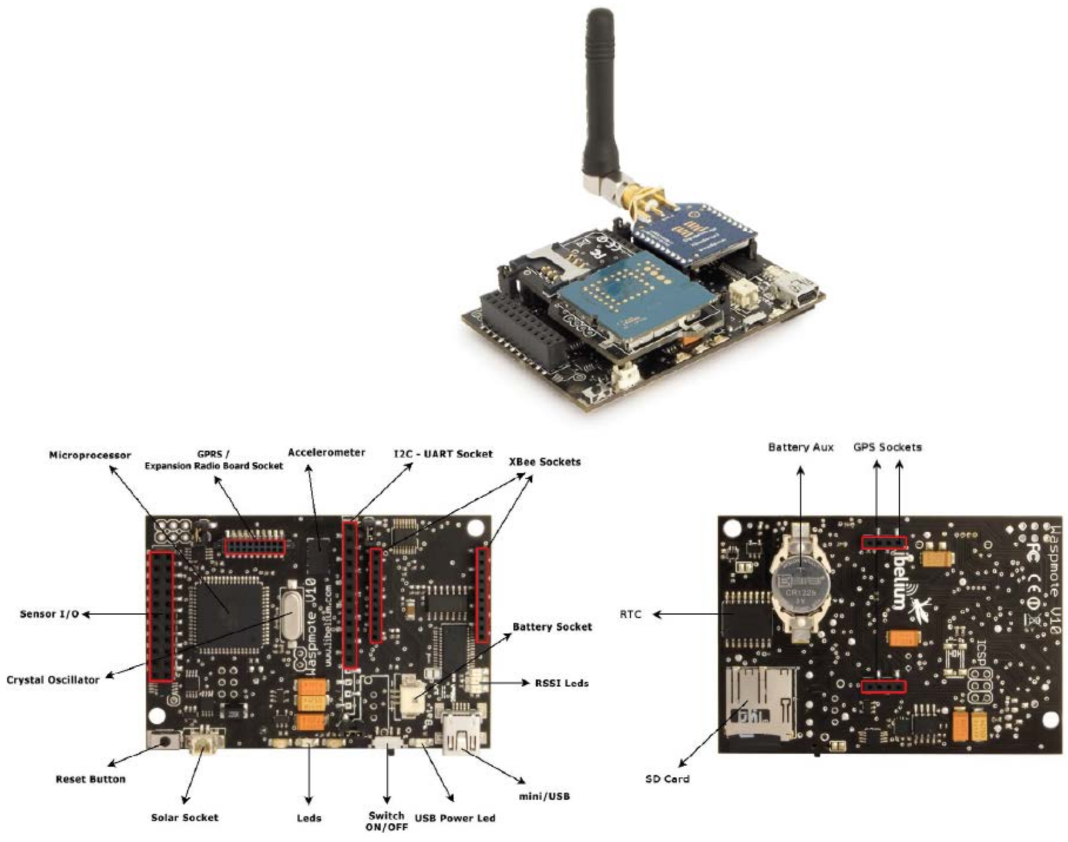


Figure 29: a Waspmote with its modules and top and bottom views of it [Libelium 12b].

A chart with its most relevant technical features has been placed below; it has to be born in mind that Waspmotes are conceived as *modular*: the subsystems used for radio transmission or data detection can be interchanged and up/downgraded. Therefore, depending on the subsystem mounted, great variations in output and performance may spring up.

Central Processing Unit		**Interfaces**	
Microcontroller	Atmel ATmega1281	Digital	8 GPIO
RAM	8 KB	Analogical	7 ADC
ROM	128 KB (Flash) + 4 EEPROM	Serial	I^2C, USB, PWM, 2 X UART
Speed	8 MHz	Sensors	Accelerometer and temperature; specific sensor boards available
Operating System	Not defined.	Gateway Infrastructure	Waspmote Gateway (to PC)
Radiofrequency		**Others**	
Transceiver	Xbee modules (manufactured by Digi)	Battery	Battery 3.3 V - 4.2V, Auxiliar battery of 3 V.
Band	868, 900 MHz, 2.4 GHz (depending on module)	Consumption (idle)	0.7 uA (hibernate mode)
Data speed	38.4 Kbps	Consumption (Rx/Tx)	9 mA
Modulation	AFH (79 channels, 1MHz bandwidth each)	Price	From 130 to 233 € per unit (depending on module)
Range	500 meters -12 Km. outdoor (depending on module).	Gateway	None
Protocol	802.15.4, ZigBee-Pro, Radiofrequency	License	GNU
Security	Proper of 802.15.4, ZigBee-Pro. AES 128 for data ciphering.	Extras	2 Gb SD card

Chart 4: Waspmote technical features.

2.4.1.4 MEMSIC.

Crossbow Technologies was once the leading marketer on the field of motes and devices for Wireless Sensor Networks, due to its widespread variety of products and solutions; however, on 3rd June 2011 it was purchased by another company called Moog Inc. to form the conglomerate Moog Crossbow Inc.

and let the former company "*focus on its high-performance inertial products for defence and security*" [Moog-Crossbow 12]. Their former Wireless Sensor Networks commercial lines where sold during 2009 and 2010 to MEMSIC Inc, the company that has retaken the efforts of development in this field.

Figure 30: left, Moog Crossbow logo [Moog-Crossbow 12], right, MEMSIC logo [MEMSIC 10].

Given these circumstances, it should come not as a surprise that MEMSIC products are almost completely equal as the motes that Crossbow Technologies had, with the exception of the Lotus platform, which is somehow an evolution of Crossbows Imote2, among others (Lotus is quoted as "*an advanced wireless node platform developed around the low power ARM7 Cortex M3 CPU and incorporates the best of IRIS, TelosB and Imote2 onto a single board*" [MEMSIC 10]).

2.4.1.4.1 Lotus

Lotus is the more advanced development of MEMSIC. It is supposed to have been built on a stackable, modular design, even with connectors in case expansion boards are required. Many of its features are related to the former Crossbow´s Imote2 platform.

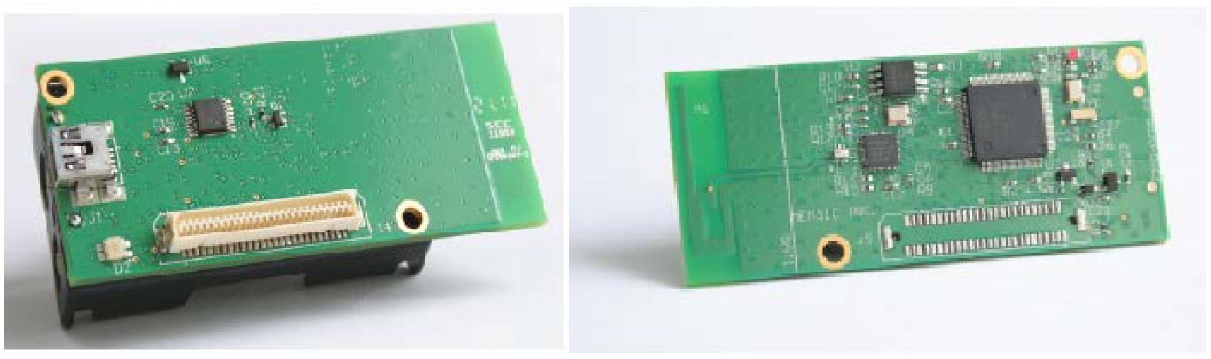


Figure 31: top (left) and bottom (right) views of Lotus mote main module [Lotus 12].

A chart with the most important data regarding this mote is below:

Central Processing Unit		Interfaces	
Microcontroller	ARM 7 Cortex M3 32-bit	Digital	GPIO
RAM	64 KB	Analogical	ADC

ROM	512 KB (Flash) + 64 Mb serial Flash	Serial	I²C, I²S, SPI, 3 X UART
Speed	10-100 MHz	Sensors	None in the platform, expansion connector for Light, Temperature, RH, Barometric Pressure, Acceleration/Seismic, Acoustic, Magnetic and other MEMSIC Sensor Boards.
Operating System	RTOS (Real Time Operating System), other options available are: MEMSIC Kiel, RTOS, IAR Systems, Free RTOS, MoteRunnerTM and TinyOS	Gateway Infrastructure	Not mentioned.
Radiofrequency		**Others**	
Transceiver	Atmel RF231 IEEE 802.15.4 Compliant	Battery	2 X AA Battery, external power 2.7 V - 3.3 V.
Band	2.4 GHz	Consumption (idle)	10 uA
Data speed	250 Kbps	Consumption (Rx/Tx)	50 mA
Modulation	ISM Band (1MHz steps)	Price	Not mentioned, former Imote2, 404 $ per unit.
Range	100 meters (outdoor).	Gateway	None
Protocol	802.15.4.	License	Not mentioned.
Security	Proper of 802.15.4.	Extras	Not mentioned.

Chart 5: Lotus technical features.

2.4.1.4.2 Iris

When this platform first came out, it was claimed to be aimed to low-powered Wireless Sensor Networks above other applications. MEMSIC offers this mote either as a lone platform or a mote with sensor boards already mounted. Compared to the previous MICA Motes, it offers twice as much program memory and three times as much radio range.

Since the acquisition of all former Crossbow´s developments, MEMSIC is delivering IRIS motes with a new software platform invented by IBM computer scientists in Zurich called Mote Runner.

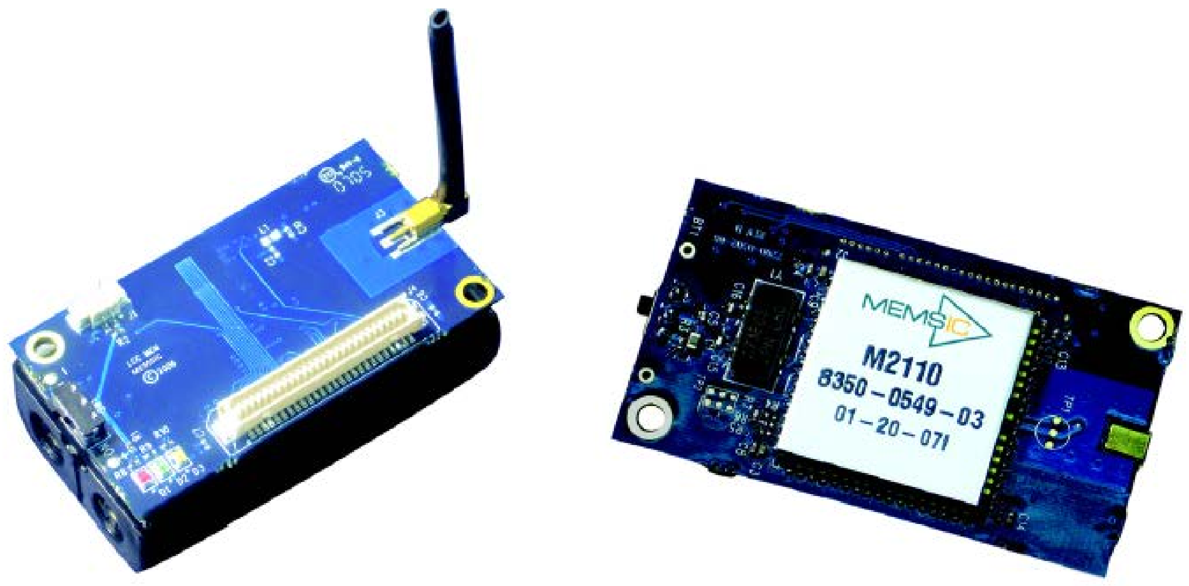


Figure 32: top (left) and bottom (right) views of Iris mote [Iris 12].

The most relevant data of Iris motes appear in the following chart:

Central Processing Unit		**Interfaces**	
Microcontroller	Amtel ATmega1281	Digital	DIO
RAM	8KB	Analogical	ADC
ROM	128KB (Flash)	Serial	I²C, SPI, UART
Speed	8 MHz	Sensors	None in the platform, expansion connector for Light, Temperature, RH, Barometric Pressure, Acceleration/Seismic, Acoustic, Magnetic and other MEMSIC Sensor Boards.
Operating System	TinyOS	Gateway Infrastructure	Not mentioned.
Radiofrequency		**Others**	
Transceiver	XM2110CB	Battery	2 X AA Battery, external power 2.7 V - 3.3 V.
Band	2405 MHz -2480 MHz	Consumption (idle)	8 uA
Data speed	250 Kbps	Consumption (Rx/Tx)	16 mA
Modulation	ISM Band (1MHz steps)	Price	155 $ per unit.

Range	>300 metres (outdoor), >50 metres (indoor)	Gateway	None
Protocol	802.15.4.	License	Open-source.
Security	Proper of 802.15.4.	Extras	Measurement 512K bytes serial Flash

Chart 6: Iris mote technical features.

2.4.1.4.2 MICAz/MICA2

These are a third generation development of the motes that were manufactured by Crossbow Technologies. Actually, they were the first motes ever manufactured and marketed by the company. Their hardware and components are basically the same; the only difference is that MICA2´s radio module consumes less energy when receiving a radio signal. MICAz and MICA2 motes are also sold as base stations: motes without batteries that are plugged to a PC via an interface (usually a USB one) and that may be used to send commands from the PCs themselves. This is a kind of product that other manufacturers also commercialize (like Oracle) and it will come in handy for our own ubiquitous system.

MICAz mote looks like this:

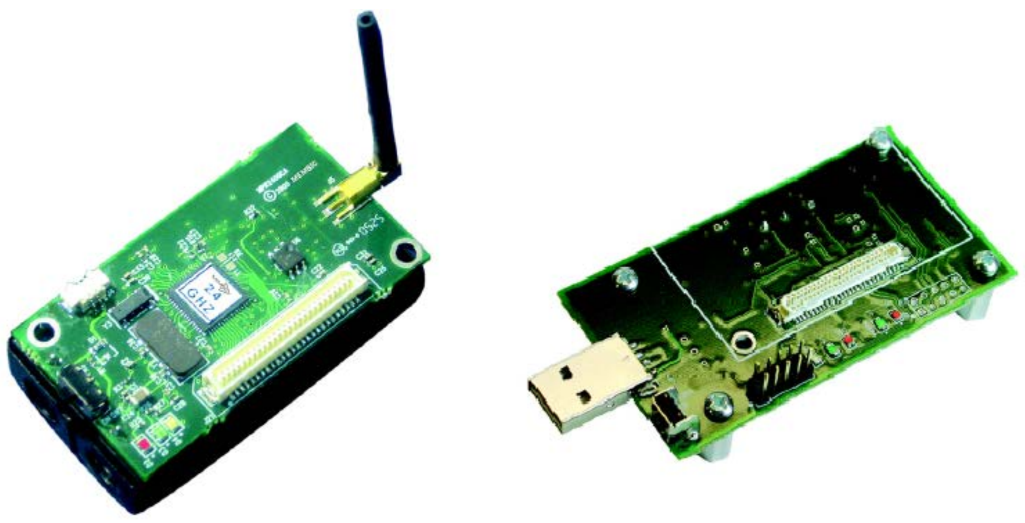


Figure 33: top view of a MICAz mote (left) its base station MIB520CB counterpart (right) [MICAz 12].

Whereas MICA2 mote, along with its twin base station has a slightly different appearance:

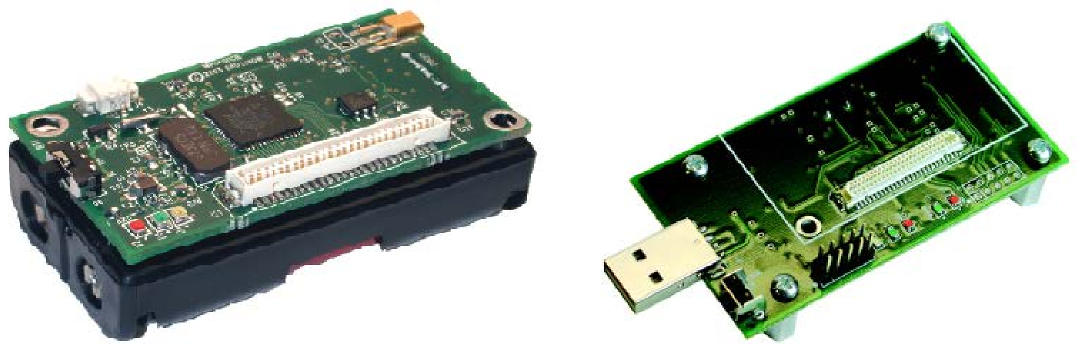


Figure 34: top view of a MICA2 mote (left) its base station MIB520 counterpart (right) [MICA2 12].

A chart comprising all the most relevant features of MICAz mote can be seen below:

Central Processing Unit		**Interfaces**	
Microcontroller	Amtel ATmega128L	Digital	DIO
RAM	4KB	Analogical	8 X ADC
ROM	128KB (Flash)	Serial	I^2C, SPI, 2 X UART
Speed	7.37 MHz	Sensors	None in the platform, expansion connector for Light, Temperature, RH, Barometric Pressure, Acceleration/Seismic, Acoustic, Magnetic and other MEMSIC Sensor Boards.
Operating System	TinyOS	Gateway Infrastructure	USB/Serial for local PC, Ethernet for base stations.
Radiofrequency		**Others**	
Transceiver	CC2420	Battery	2 X AA Battery, external power 2.7 V - 3.3 V.
Band	2.4 GHz	Consumption (idle)	16 uA
Data speed	250 Kbps	Consumption (Rx/Tx)	27,7mA/25,4mA
Modulation	O-QPSK	Price	134 $ per unit.
Range	75-100 metres (outdoor), 25-30 metres (indoor)	Gateway	None
Protocol	802.15.4.	License	BSD (operating system).

Security	Proper of 802.15.4. AES 128 for data ciphering	Extras	Measurement 512K bytes serial Flash

Chart 7: MICAz mote technical features.

Whereas MICA2 mote most relevant technical data is displayed here:

Central Processing Unit		**Interfaces**	
Microcontroller	Amtel ATmega128L	Digital	DIO
RAM	4KB	Analogical	8 X ADC
ROM	128KB (Flash)	Serial	I²C, SPI, 2 X UART
Speed	7.37 MHz	Sensors	None in the platform, expansion connector for Light, Temperature, RH, Barometric Pressure, Acceleration/Seismic, Acoustic, Magnetic and other MEMSIC Sensor Boards.
Operating System	TinyOS	Gateway Infrastructure	USB/Serial for local PC, Ethernet for base stations.
Radiofrequency		**Others**	
Transceiver	CC1000	Battery	2 X AA Battery, external power 2.7 V - 3.3 V.
Band	868/916MHz (MPR400CB), 433MHz (MPR410CB), 315 MHz (MPR420CB)	Consumption (idle)	16 uA
Data speed	38.4 Kbps	Consumption (Rx/Tx)	18 mA/35 mA
Modulation	FSK	Price	155 $ per unit.
Range	Outdoors: 152.40m (MPR400), 304.8m (MPR410 y MPR420)	Gateway	None
Protocol	Not defined.	License	BSD (operating system).
Security	Not defined.	Extras	Measurement 512K bytes serial Flash

Chart 8: MICA2 mote technical features.

2.4.1.4.3 MICA2dot

This model of mote is essentially a MICA2 mote dramatically shrunk to the size of a quarter-dollar coin (25 mm diameter); due to that feature, its energy consumption is lower than its bigger counterpart. Mica2dot is expected to be useful in environments where especially non-intrusive measurement devices are required, or as it is said by MEMSIC, “*Deeply Embedded Wireless Sensor Networks*”.

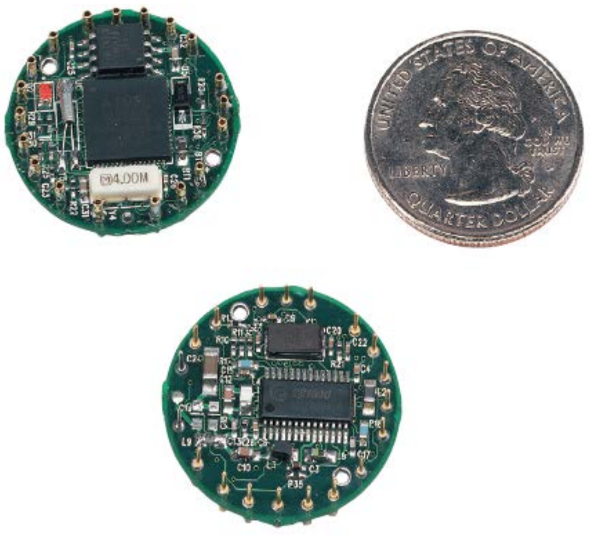


Figure 35: top and bottom views of a MICA2dot, compared with a quarter dollar coin [MICA2Dot 12].

A chart with its most relevant technical data is as follows:

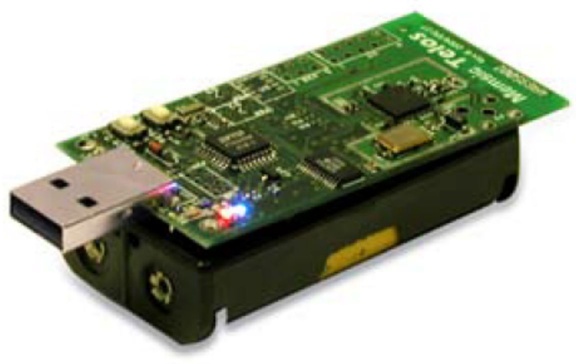
Central Processing Unit		**Interfaces**	
Microcontroller	Amtel ATmega128L	Digital	DIO
RAM	4KB	Analogical	8 X ADC
ROM	128KB (Flash)	Serial	I²C, SPI, 2 X UART
Speed	7.37 MHz	Sensors	None in the platform, expansion connector for Light, Temperature, RH, Barometric Pressure, Acceleration/Seismic, Acoustic, Magnetic and other MEMSIC Sensor Boards.
Operating System	TinyOS	Gateway Infrastructure	USB/Serial for local PC, Ethernet for base stations.
Radiofrequency		**Others**	
Transceiver	CC1000	Battery	Low-Mass battery.

Band	868/916MHz, 433MHz, 315 MHz	Consumption (idle)	<15 uA
Data speed	38.4 Kbps	Consumption (Rx/Tx)	8 mA
Modulation	FSK	Price	142 $ per unit.
Range	Outdoors: 152.40m-304.8m	Gateway	None
Protocol	Not defined.	License	BSD (operating system).
Security	Not defined.	Extras	Measurement 512K bytes serial Flash

Chart 9: MICA2dot mote technical features.

2.4.1.4.4 TelosB

TelosB is an open-source platform especially designed for research and development in the community of the Internet of Things. In order to enlarge its target to this kind of consumer, TelosB mote is marketed with several characteristics to make development easier: a USB port for uploading/downloading code to/from the mote or TinyOS as an open-source operating system.



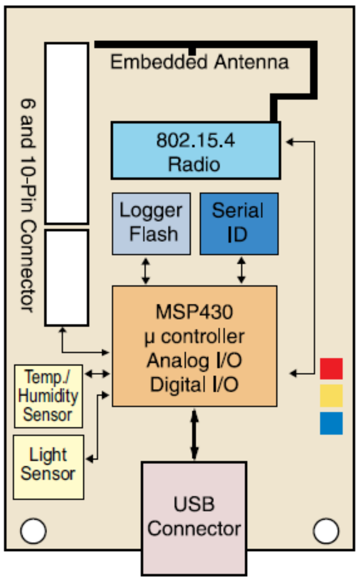


Figure 36: top view of a TelosB mote and a block diagram [TelosB 12].

Its most interesting technical features can be observed here:

Central Processing Unit		Interfaces	
Microcontroller	TI MSP430	Digital	2 X DAC

RAM	10KB	Analogical	ADC
ROM	48KB (Flash)	Serial	I^2C, SPI, UART
Speed	8 MHz	Sensors	Integrated Temperature, Light and Humidity Sensor
Operating System	TinyOS	Gateway Infrastructure	USB for local PC
Radiofrequency		**Others**	
Transceiver	CC2420	Battery	2 X AA.
Band	2,4 MHz -2483,5 MHz	Consumption (idle)	5.1 uA
Data speed	250 Kbps	Consumption (Rx/Tx)	1.8 mA
Modulation	ISM band	Price	174 $ per unit.
Range	75-100 metres (outdoors), 20-30 metres (indoors)	Gateway	None
Protocol	802.15.4.	License	Open source.
Security	Proper of 802.15.4.	Extras	Not mentioned.

Chart 10: TelosB mote technical features.

2.4.1.4.5 MCS Cricket

This mote has a very unique characteristic among MEMSIC commercial line of motes: it features an ultrasound transmitter/receiver that can be used, for example, to measure the difference of arrival time between two signals, when the ultrasound transmitter/receiver is combined with the regular radiofrequency technology. Other than that, their hardware is pretty much that of the MICA2´s, as it can be seen on the next page:

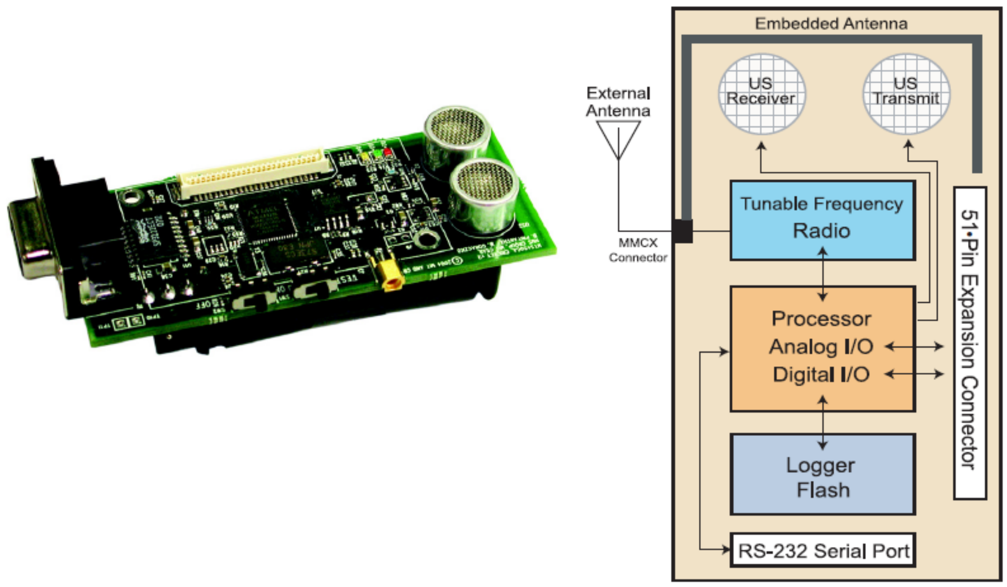


Figure 37: top view of a MCS Cricket mote and its block diagram [Cricket 12].

Their most important technical features are:

Central Processing Unit		**Interfaces**	
Microcontroller	Amtel AVR	Digital	DIO
RAM	4KB	Analogical	8 X ADC
ROM	128KB (Flash)	Serial	I^2C, SPI, 2 X UART
Speed	7.37 MHz	Sensors	None in the platform, expansion connector for Light, Temperature, RH, Barometric Pressure, Acceleration/Seismic, Acoustic, Magnetic and other MEMSIC Sensor Boards.
Operating System	TinyOS	Gateway Infrastructure	Serial DB9.
Radiofrequency		**Others**	
Transceiver	CC1000	Battery	2 X AA Battery, external power 2.7 V - 3.3 V.
Band	868/916MHz (MPR400CB), 433MHz (MPR410CB), 315 MHz (MPR420CB)	Consumption (idle)	16 uA
Data speed	38.4 Kbps	Consumption (Rx/Tx)	18 mA/35 mA

Modulation	FSK	Price	263 $ per unit.
Range	Outdoors: 152.40m (MPR400), 304.8m (MPR410 y MPR420)	Gateway	None
Protocol	Not defined.	License	BSD (operating system).
Security	Not defined.	Extras	Measurement 512K bytes serial Flash

Chart 11: MCS Cricket mote technical features.

2.4.1.5 Oracle.

Although Oracle was traditionally known for its software and certification businesses, especially the ones around databases, in 2010 they purchased the Java creator and developer Sun Microsystems, which had been doing developments in the field of motes for a time (Sun SPOT model). Therefore, now it has to be considered as another manufacturer with an interest for the scope of this project [Oracle 12].

Figure 38: Oracle corporation logo with the Sun logo above it [Oracle 12].

2.4.1.5.1 Sun SPOT

The importance of Sun SPOT (Sun Small Programmable Object Technology) motes is huge in this Final Degree Dissertation, as they are the model of motes that has been put to a use in our model for a ubiquitous system. Sun SPOT motes are 802.15.4 compliant and, as a unique feature among most motes, it uses a downsized version of the Java Virtual Machine typical of Java developments, called Squawk, thus using Java 2 Micro Edition as the programming language of choice. This reduced virtual machine will work as the mote operating system as well. Apart from the battery-powered motes, development kits are sold with base station motes, used for communication purposes from USB-capable devices (Sun SPOT motes use mini-USB as a gateway with the outside) to battery-powered motes.

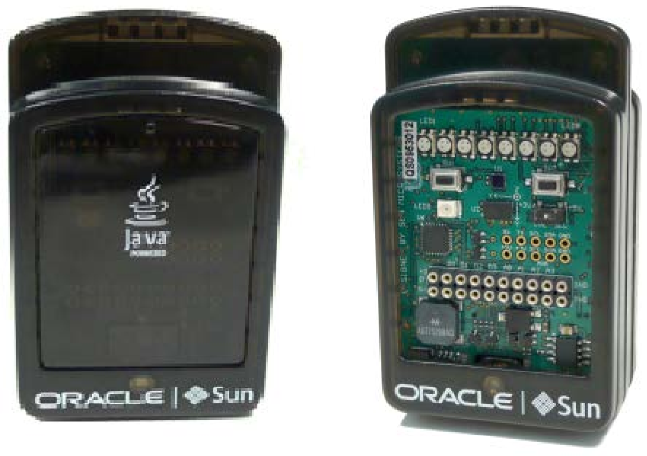


Figure 39: a Sun SPOT mote with its safety lid on (left), without it (right) [Sun SPOT 12].

Its most relevant technical features are as follows [Sun SPOT 12]. It can be observed that some of them, especially regarding RAM and ROM values are particularly impressive, at the cost of having an energy consumption rate a little higher than some of the other motes:

Central Processing Unit		**Interfaces**	
Microcontroller	AT91SAM9G20	Digital	GPIO
RAM	1Mbyte	Analogical	ADC
ROM	8Mbytes (Flash)	Serial	I²C, SPI, UART
Speed	400 MHz	Sensors	Light, Temperature, Accelerometer
Operating System	None, Squawk virtual machine performs that role.	Gateway Infrastructure	Mini USB for upload/download of code and communication.
Radiofrequency		**Others**	
Transceiver	CC1000	Battery	770mAhr Li-Ion Rechargeable Battery
Band	868/916MHz (MPR400CB), 433MHz (MPR410CB), 315 MHz (MPR420CB)	Consumption (idle)	30 uA
Data speed	38.4 Kbps	Consumption (Rx/Tx)	105.6 mW
Modulation	FSK	Price	630 € per kit (two motes and a base station).
Range	Outdoors: 152.40m (MPR400), 304.8m (MPR410 y MPR420)	Gateway	None

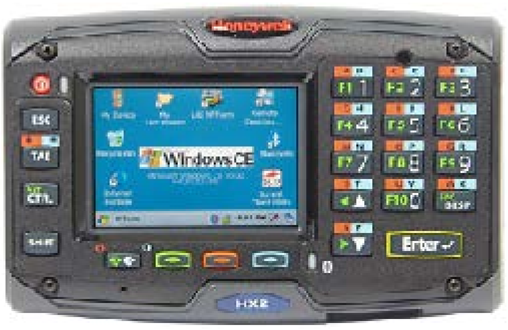
Protocol	Not defined.	License	Open source.
Security	Not defined.	Extras	None

Chart 12: Sun SPOT mote technical features.

2.4.1.6 Other kinds of ubiquitous hardware.

Given that motes and elements from Wireless Sensor Networks are the most mature and notorious among ubiquitous computation, other pieces of equipment have to be considered that, while are not common as part of a Wireless Sensor Network, can easily be regarded as belonging to a Internet of the Future infrastructure; usually, they are in the place close to a end user, or are taking part in a wireless communication in different terms.

Wearable computers focused on the final user are a field even more heterogeneous than motes. In this case, by wearable computers it is meant computers that have been designed to be worn as if they were a non-electronic device, such as a piece of garment or a bracelet. This kind of computer is already in use nowadays, although not always with ubiquitous purposes. Nevertheless, the fields where they are employed (logistics, body monitoring) can be ported to a ubiquitous stance with ease. Unlike motes or devices that interact with others, these wearable computers must offer a machine-to-human interface for data to be understood, like a screen or a low-capacity printer, and a human-to-machine interface in order to input data (like the amount of a set of products on an inventory, etc.). Vendors like Honeywell [Honeywell 12] or Sony [Yanko 10] are manufacturing their own devices.



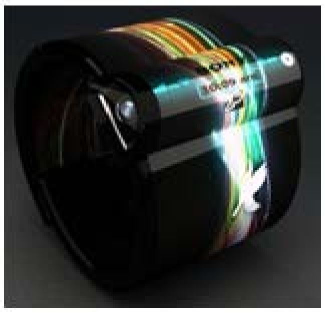


Figure 40: Honeywell HX2 device (left), and Sony Nextep prototype.

Despite their small dimensions, it is often considered that a **mobile phone** is not bond to be a wearable computer *per se*, because they were not designed to have functionalities of a wearable computer designed for a pervasive, ubiquitous environment and in their manufacturing processes it is very unlikely to have such objectives in mind. Nevertheless, due to their features, they can be used as a final user wearable computer if applications related to specific scenarios are codified and there is a human user who can take advantage of them in a ubiquitous environment. Larger electronic appliances, like laptops or netbooks, are less likely to be part of a scenario with these features as wearable computers.

Finally, it should be mentioned that in wearable computing, one of the most daring proposals is **electronically augmented clothes**. Rather than having a machine-to-human interface in order to inform about data (usually physical data, like body temperature or skin sweat levels), cases where electronics integrated in clothing will notify this data to a monitoring entity (like a hospital or a fire station), ideally in real time, are the most dominant tendency. Thus, a much more efficient effort can be put in taking care of the final user of the electronic apparel [Northeastern 10].

Figure 41: data logging shirt [Northeastern 10].

2.4.2 Embedded software in hardware platforms.

When referring to this kind of software, it is meant to say the built-in software that will be found in the devices used for ubiquitous systems, especially motes belonging to a Wireless Sensor Network. It will usually be an operating system, as the ones that have already been reviewed before or, in the case of Sun SPOT motes, a middleware platform (Squawk Virtual machine) that will take over the functionalities expected from an operating system. Often, though, the line delimiting the functionalities from one layer to other (operating system, middleware, application) are not entirely precise; cross-layer optimization could be taken in order to optimize the overall performance of the system [Vicente 10], despite somewhat breaking with the idea of having a solution as a stack of separated layers. As in all the other features, operating systems and embedded software in these platforms especially prepared for the special working conditions of a ubiquitous, pervasive system.

One of the most relevant operating system for Wireless Sensor Networks devices is **Contiki**. This is a C-codified operating system for memory-constrained embedded networks systems, such as our Wireless Sensor Networks case, that is used in many low-power processors and devices. It was designed by Alam Dunkels in the Swedish Institute of Computer Science (SICS). Contiki is a highly portable operating system that runs under an open source BSD-like license that allows it to be freely downloaded. It offers concurrence support, along with IPv4 and IPv6 support, and low-power standards such as 6loWPAN, RPL or CoAP as well [Contiki 12].

Swedish Institute of Computer Science SICS

Figure 42: SICS logo.

In a hardware system, typical memory requirements for a Contiki implementation are 2Kb of RAM and 40 Kb of ROM. Contiki makes use of a event-directed kernel, where programs and applications are uploaded and downloaded dynamically in real time. In order to tackle concurrence issues, Contiki will employ processes that are using *protothreads* (very light threads) on the previously quoted kernel. Contiki will communicate one process to another by means of events and a subsystem with a user graphic interface for interconnected terminals, locally or via network (Telnet, VNC´s virtual display).

In order to guarantee interconnectivity between conventional networks (either in a wireless environment or not) and pervasive, ubiquitous ones, Contiki makes use of two different communication stacks, uIP and Rime. uIP is used for processes related with TCP/IP architecture (it can even communicate with UDP-enabled entities) and therefore is able to communicate with other machines throughout the Internet. Rime, on the other hand, is used for low-powered networks, providing varied communication possibilities, from local broadcasts with best-effort quality of service to running over uIP and mapping all the communications below to a ubiquitous, pervasive environment [Österlind 09].

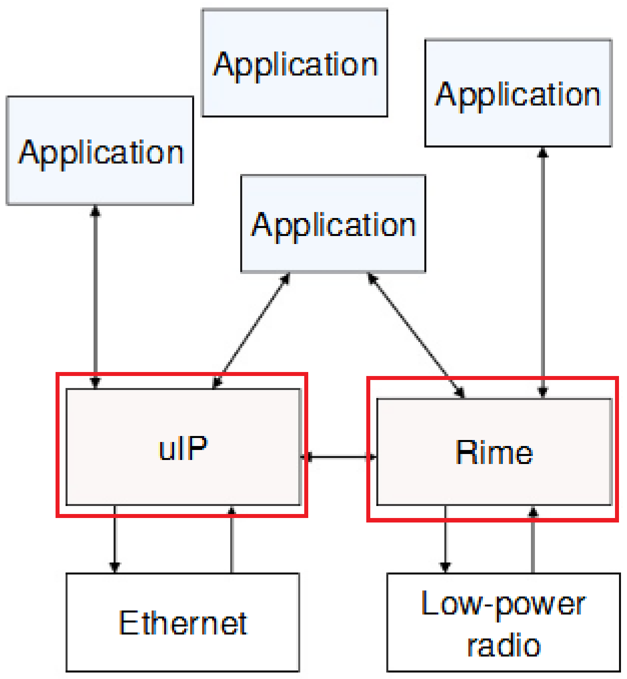


Figure 43: Contiki´s communication stacks and how they interrelate [Österlind 09].

As another example of how software is embedded on low capabilities platforms the operating system of BTNode motes, that is **BTNut**, can be introduced. In fact, BTnut consists of three major components: a general purpose operating system called Nut/OS, able to communicate with a TCP/IP stack (and most of its related user protocols: HTTP, UDP, ICMP, ARP...), specific drivers for the BTnode mote, and the

Bluetooth stack for Bluetooth communications [BTNode 09]. BTnut can be compiled from a native UNIX target, and provides APIs for its most relevant modules: Nut/OS, Nut/Net (for network communications), FileSystem, Device drivers, runtime library and even offers support for Nut/OS mods. As Contiki, BTNut is written using C as the programming language.

Perhaps the most widely known operating system for low-capabilitiy devices is **TinyOS**. As others, TinyOS is an open-source, component-based operating system focused on the development of software and hardware targeting Wireless Sensor Networks above all other electronic systems, and has been designed as such from its very beginning. The evolution and support for TinyOS seems assured as there is a consortium, the TinyOS Alliance, with a strong interest in its development.

Figure 44: TinyOS logo.

Instead of making use of multithreading, TinyOS is based on event-driven programming, which is a much more useful approach given the ever-shifting nature of Wireless Sensor Netowrks, often disturbed by events (new nodes, node failures, etc.). It is an operating system written in the C-based programming language nesC (network embedded systems C); this language is also used by application developers for programming software on TinyOS. TinyOS has a wide library of components including sensor handlers, network protocols, distributed services, and data acquisition tools than can be used either as they are by default or modified to best suit the developers.

As it was mentioned before, TinyOS has a modular structure [Familiar 09]. Each of the modules can be broken into three different mandatory parts (they can be void in a component implementation but must be present in any component to be TinyOS compliant):

- **Configuration**. This part will always be present, but unless the component is made by composition of other previously created components, will be void.
- **Implementation**. This part will define the connections between the different components used by the application.
- **Module**, the largest section because the application behaviour is programmed here. It is subdivided into three more parts: *uses* (interfaces used by the component), *provide* (interface supplied by the component) and *implementation* (business logic of the component).

On the other hand, using nesC as the programming language seems like a wise choice because it is partially based on Object-Oriented Programming: it has an event handler proper of languages as Visual

Basic, it is interface and event-based, and it still resembles C (the programming language most widely used in embedded systems) enough to get familiar with it.

It is common for a TinyOS application to take control of the whole system it is installed upon. Here, some of the functionalities expected from one layer may be tackled by other one for the good of the application, then making use of the principle of cross-layer optimization. In this way, the memory required to store the code will be smaller than expected, but a complete image will be required to be uploaded to the mote every time it has to be reprogrammed.

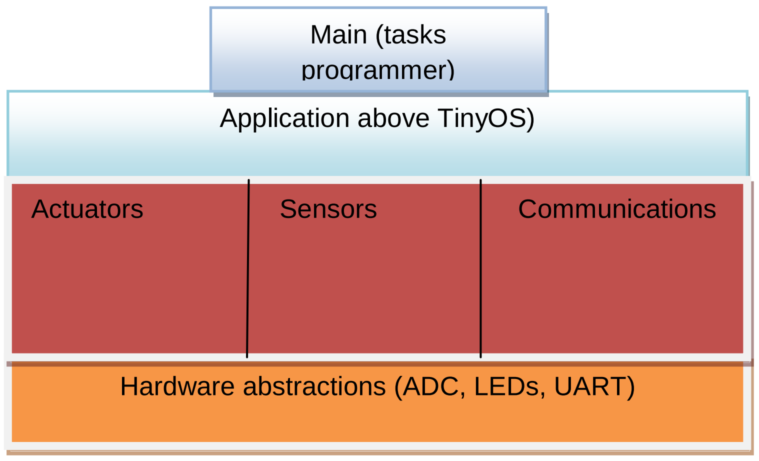

Main (tasks programmer)
Application above TinyOS)
Actuators
Sensors
Communications
Hardware abstractions (ADC, LEDs, UART)

Figure 45: application structure with TinyOS below; red and orange layers are TinyOS-based.

Finally, a mention should be done to Oracle´s **Squawk Virtual Machine**. It is mostly written in Java code, unlike all the other motes and middleware architectures, usually codified in C/C++ or in lightweight versions of these languages. Due to this, a seamless integration with objects programmed under an Object-Oriented Paradigm is done in a more comfortable way, especially if they are Java-based. What is more, Mobile Editions of Java (and particularly, J2ME) have a very small memory footprint and memory usage can be kept to a minimal. As a positive result, Squawk allows one or more applications to run over the same Virtual Machine, keeping them isolated one from the other.

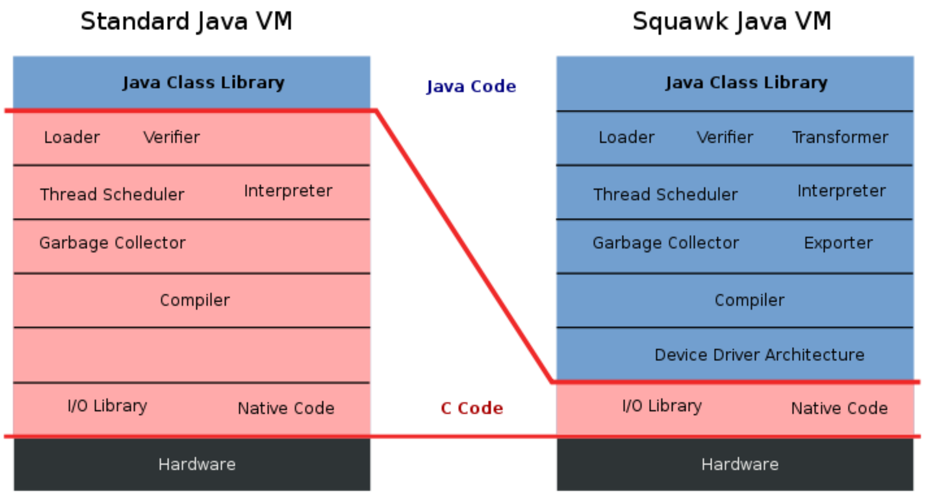


Figure 46: comparison between a Java VM and the Squawk VM [JVM 12].

2.4.3 Open issues.

As it can be figured out, the main issues about the hardware platforms shown before are gravitating around the size and the cost of the nodes. Ideally, the smaller the node, the more advantageous is its usage: a lesser amount of materials are used for its manufacturing, they introduce less disturbances in the scenarios they are placed in, and they become more pervasive and ubiquitous for a final user perspective. However, as it has been said before, downsizing the used devices implies a huge investment in Research and Development and in human resources, so although considerable downsizing is expected to happen sooner than later, it will not be something that will happen suddenly, but as a gradual effort. As far as pricing and costs are concerned, as soon as the Internet of the Future becomes as widespread as the “regular” Internet, mass-production of motes and end user wearable computers, while already taking place, will be intensified in the mid-term future and therefore prices should decrease, although it is not expected that it will be done right away; rather, new developments, with improved characteristics (more RAM/ROM memory, interfaces, etc.) and even more shrunk in size will appear at usually mid-to-high pricing, thus forcing the former state-of-the-art devices of the market to drop their prices.

Other concern about hardware platforms is their energy management. Since one of the main objectives of the Internet of the Future is having electronic devices as seamlessly integrated as possible, any piece of equipment that has to be frequently charged will not be that transparent to the final user. Currently there are two different directions to solve this problem:

- **Extending node usable time before recharging**. Here, research is focused on improving the lifetime of the battery, either when the power depends on AA batteries or Li-Ion rechargeable ones, and reducing the power consumption of the node itself (and while they are on it, giving feedback on the miniaturization efforts tackled before).

- **Providing means of self-recharging the nodes**. Energy harvesting is one of the top research topics as well; currently there are some ideas that are being shyly implemented, like solar-powered Wireless Sensor Networks [Yastrebova&07]. Furthermore, kinetic and mechanical energy generated by objects under movement (for example, a bridge or a city structure expected to have certain quantities of vibration) or thermal currents generated by the difference of temperature between two junctions of conducting material can be used in energy harvesting as well [Seah&09]. Even trees can generate electricity that can be used for recharging motes from pH differentials between the tree and the soil [de Castro 11].

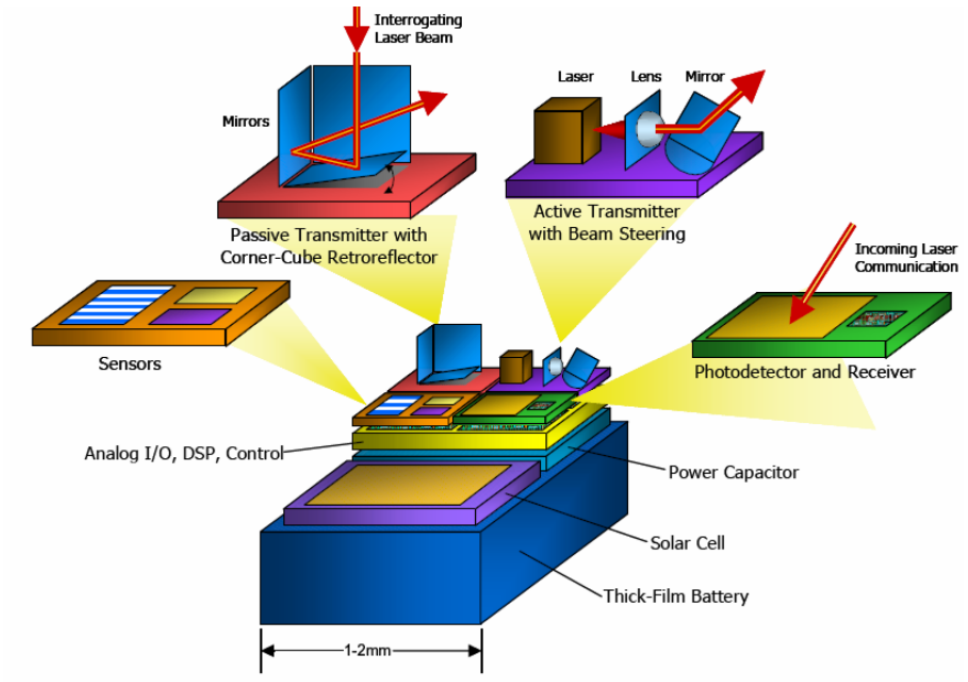

Interrogating Laser Beam
Mirrors
Passive Transmitter with Corner-Cube Retroreflector
Laser
Lens
Mirror
Active Transmitter with Beam Steering
Incoming Laser Communication
Sensors
Photodetector and Receiver
Analog I/O, DSP, Control
Power Capacitor
Solar Cell
Thick-Film Battery
1-2mm

Figure 47: components of a solar energy harvesting mote [Yastrebova&07].

Another complain that can be raised is regarding the radio frequency systems in motes. Wireless Sensor networks need antennae able to receive signals from uncorrelated (not varying together at the same time and quantity) sources; an efficient and normalized protocol would be very useful to ensure an expectable behaviour and improved data transfer rates for the different applications. Radio communication protocols in a Wireless Sensor Network should be improved too, though.

Finally, security and Quality of Service capabilities should be improved in order to become at least comparable with a wired or Wi-Fi system, as it was pointed out when reviewing the open issues of middleware architectures. In particular, fixing certain Quality of Service for ubiquitous networks as a standard could be really beneficial, for these systems are usually more prone to errors and failures, and it would become a way to make them compliant with minimum functionalities.

Chapter 3

Service and wearable-oriented semantic middleware: nSOM

3.1 nSOM architecture: an introduction.

nano Service-Oriented Middleware (hereinafter, nSOM) is a semantic middleware architecture developed by several researchers at the GRyS (Grupo de Redes y Servicios de próxima generación) group, belonging to the EUITT (Escuela Universitaria de Ingeniería Técnica de Telecomunicación) from the UPM (Universidad Politécnica de Madrid). It has been designed to be used in a ubiquitous environment, especially as a middleware for a Wireless Sensor Network. It takes into account the latest ideas in ubiquitous computing, in order to create a middleware architecture updated enough for the purposes shown in this Final Degree Dissertation. Its general features are the following:

- It is a **middleware architecture designed for low-capability devices**. It has been created with the idea of uploading it on motes part of a Wireless Sensor Network, so any device with mote-like or higher capabilities with interconnection functionalities will be able to execute it with very little trouble

- It is a **software-centric middleware**. nSOM does not consider what devices are on the physical layer, except for its obvious middleware objectives related with hardware abstraction. Instead, it is composed by a series of modules with delimitated tasks within its architecture, as it will be seen later.

- It **makes use of agents**. nSOM communicates with and governs hardware devices by deploying agents on them. Each agent has different duties that, although usually dissimilar, can be federated to have a cooperation grid among several of them, seeking the successful delivery of a composed service. These agents are closely linked to what the place they are present (usually a mote) can offer –that is, either data collected from sensors: temperature, luminosity, etc. or the possibility to trigger an actuator, like a loudspeaker, LEDs, etc.

- It is part of a bigger framework **based on Service-Oriented Architecture and Service-Oriented Computing principles**. The main goal of nSOM is providing services to a final user; to accomplish this, it will be part of layer that can be named as the Service-Oriented Software Platform, placed in a model for service and semantic middleware with layers above and below, as it can be observed on the next figure:

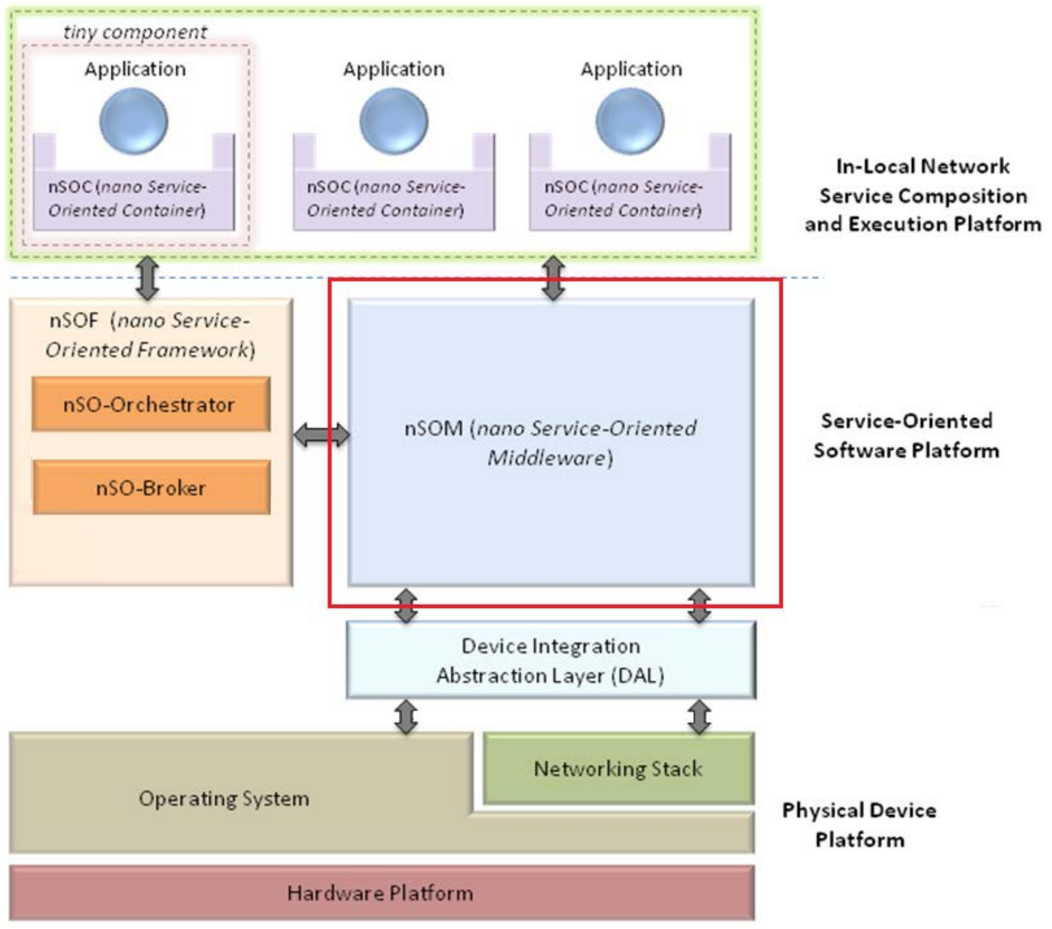

tiny component
Application
Application
Application
nSOC (nano Service-Oriented Container)
nSOC (nano Service-Oriented Container)
nSOC (nano Service-Oriented Container)
In-Local Network Service Composition and Execution Platform
nSOF (nano Service-Oriented Framework)
nSO-Orchestrator
nSO-Broker
nSOM (nano Service-Oriented Middleware)
Service-Oriented Software Platform
Device Integration Abstraction Layer (DAL)
Operating System
Networking Stack
Physical Device Platform
Hardware Platform

Figure 48: abstraction layers for service and semantic middleware [Lifewear 12].

It can be learnt from figure 48 that nSOM is included as a part in an already mentioned Service-Oriented Software Platform, along with a significant nano Service-Oriented Framework with the agents that are taking part in the services that are going to be offered, and a Device integration Abstraction Layer that is converting and homogenizing the information delivered by the hardware-based layer under it.

This latter layer (or more accurately, the Physical Device Platform) comprises all the hardware-related parts of the whole reference model: the hardware platform supporting all the higher levels, the operating system that is managing the behaviour of that hardware platform and the networking stack that is ruling the networking procedures of the hardware (that hardware will be a mote, more often than not).

On the other hand, the layer over the Service-Oriented Software Platform (In-Local Service Composition and Deployment Platform) will be taking part in giving shape to the services in use; in order to do that, tiny service building blocks (tiny as long as the capacities used are born in mind) will be devised. These tiny components will be composed by two subcomponents: a nano Service-oriented Container and the software Application contained by the former subcomponent.

Once nSOM has been located in a reference model to get a good grasp of where it is and what is expected from it, it can be studied not only from a black-boxed perspective, but also having a look at its inner components. A more specific figure with the Service-Oriented Software Platform is shown here:

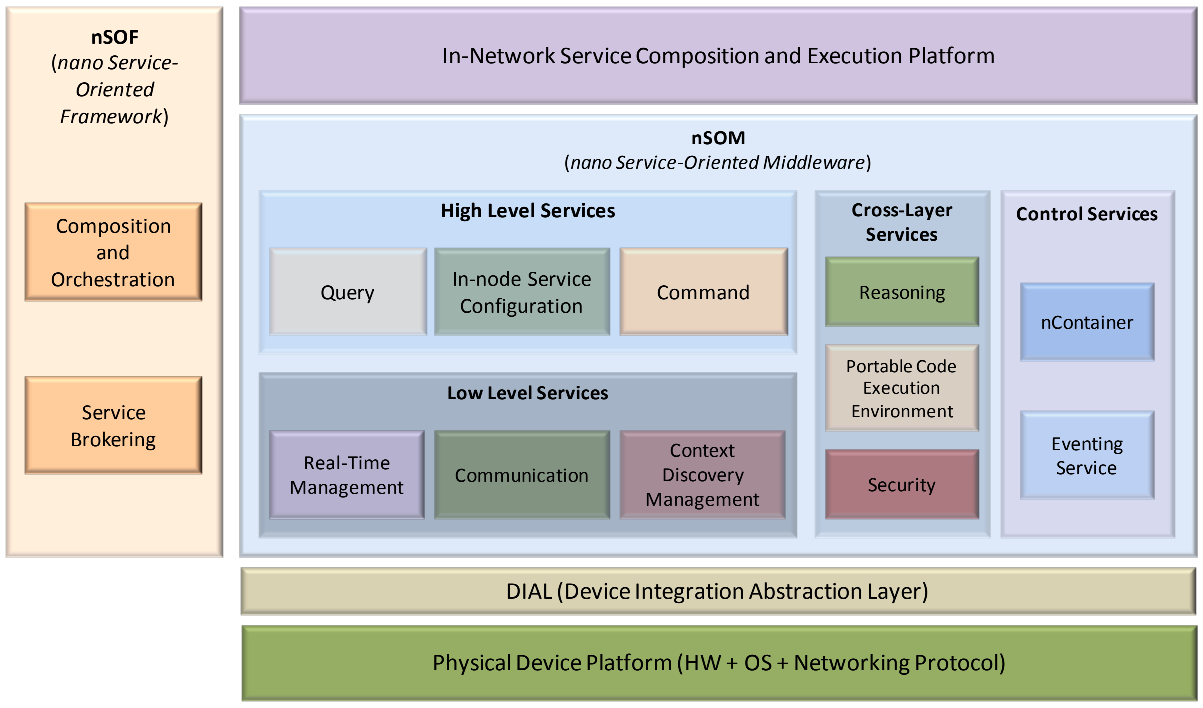

nSOF
(nano Service-Oriented Framework)
Composition and Orchestration
Service Brokering
In-Network Service Composition and Execution Platform
nSOM
(nano Service-Oriented Middleware)
High Level Services
Query
In-node Service Configuration
Command
Low Level Services
Real-Time Management
Communication
Context Discovery Management
Cross-Layer Services
Reasoning
Portable Code Execution Environment
Security
Control Services
nContainer
Eventing Service
DIAL (Device Integration Abstraction Layer)
Physical Device Platform (HW + OS + Networking Protocol)

Figure 49: nSOM components [Lifewear 12].

nSOM is done by merging four different groups of services, with their different missions and procedures:

1. **High level services**. These services are just under the In-Local Service Composition and Deployment Platform, so they can be accessed by the tiny components at that level. This access will be made from a system point of view that takes into account the needs of the In-Local Service Composition and Deployment Platform, thus making compulsory for the middleware to adapt to different conditions and giving an added value to the mote and the Wireless Sensor Network it is part of. There are three different services encased in this high level: query (in-node and inter-node management of service messages during a communication), in-node service configuration (configuration support for middleware services: acquisition context configuration, routing configuration, reasoning and knowledge management issues, security policies, aggregation functions, hardware setting-up, etc.) and command (command creation and addition possibilities for the middleware, so as to better manage tiny applications at the In-Local Service Composition and Deployment Platform).

2. **Low level services**. Services at this level are working in close cooperation with the Physical Device Platform, so consequently their functionalities are pivotal for the correct performance of the whole system. They are given tasks that are related to the processing of data obtained by the hardware devices of the Wireless Sensor Network. The services present in this

subcomponent are: real-time management (support for the configuration of the real-time capabilities available in the operating system, according to the requirements of the tiny components of the higher layer), communication (it provides an interface to abstract the functionalities typical of the network layer, as encapsulating and routing) and context discovery management (discovery capabilities and management services, capable to be used by the application layer so that it will find service providers and synchronize the business logic among the agents deployed in the ubiquitous pieces of hardware of the system).

3. **Cross-Layer Services**. Unlike the other subsystems, the services provided in this case are orthogonal, and both low level and high level services will use them to their advantage. What is more, not only the local device nSOM is mounted on will use them, but also the whole Wireless Sensor Network will benefit from them. The presented cross-layer services are three: reasoning (dynamic modification of the behaviour of the wearable device and its deployed services, according to a collection of specifications defined by an inference engine. This inference engine is the thinking part of the nSOM architecture, charged with providing global context awareness to the system under a distributed conception), portable code execution environment (support for execution of portable code from other ubiquitous pieces of hardware) and security (it defines the needed mechanisms to offer confidentiality, integrity and authentication of the data flowing through the architecture at both ends of the communication. To have it working, it is necessary to define the security services available in the WSN and the security mechanisms that will prevent attacks on the WSN, or in case an attack happens, that will reduce as much as they can the damage taken by the network).

4. **Control services.** These services are controlling the middleware architecture. They manage the deployment of the components and their lifetime cycle, as well as enabling event-based inter-component communication, thus relieving from that task other components that belong to other layers, and improving their own performance. There are two control services: nContainer (lifetime cycle management of the middleware components, offering dispatching capabilities and scheduling) and an eventing service (publishing and subscription capabilities for middleware and tiny components, taking care of dispatching and event management as well).

3.1.1 Internetworking model overview.

After describing the different components and functionalities of nSOM it would be of great use to know how all these components are deployed in a Wireless Sensor Network. It should be noted that these components can be uploaded in almost any wearable device (smart phones, etc.), but given that in the next chapter it is going to be described how a system using Sun SPOT motes has been deployed, usually the nodes of the WSN will be considered motes.

The internetworking model will be making use of a hierarchy; when there is a request from a service to be retrieved it will not be done as an *ad hoc* action; but a series of ordered messages will be sent from the user, within the motes of the WSN and the node able to provide the service. The elements present are shown in the next figure:

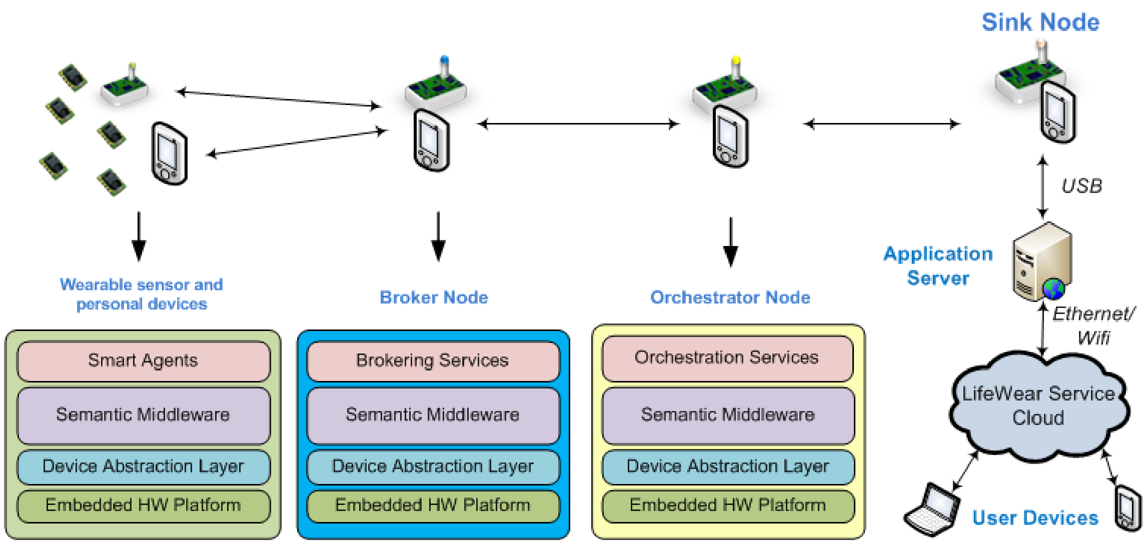


Figure 50: internetworking model [Lifewear 12].

It is of critical importance to understand that the nodes presented here (wearable sensors and personal devices, Broker, Orchestrator and Sink) are playing those roles because there are agents deployed on them (Orchestrator agent, Broker agent, etc.) that fulfil the tasks expected from their kind. What determines the functionalities of motes in a Wireless Sensor Network is not the mote, but the software uploaded on it regardless of other considerations (for example, it is possible to have several agents uploaded on a mote, but if one of the agents is the Orchestrator agent, the mote will assume the role and the responsibilities of an Orchestrator node, apart from whatever tasks are associated to the other deployed agents on the mote).

The agents and their functionalities are the next topic this Final Degree Dissertation is going to deal with.

3.1.1.1 Base station/ Gateway/ Sink.

The base station plays a role similar to a gateway in other kinds of networks. Effectively, it will be the bridge between the Wireless Sensor Network and the conventional network. Under a more physical point of view, the Gateway (also named as Base station and Sink) will be sharing features from the two worlds: it will usually be a ubiquitous device capable to interact with a non-ubiquitous device in charge of processing the data retrieved from the Wireless Sensor Network; as an example, the device used in the scenario that will be introduced in Chapter 4 is a Sun SPOT base station, a special mote that, while is able to communicate with all the other deployed motes, it also makes use of the code that is located in a PC, and is attending requests coming from an ESB deployed at the same Personal Computer. In this case, the base station is plugged to the PC via a USB interface, and since it has no power on its own, must remain connected all the time to attend requests.

This is a unique case in the nSOM architecture in the sense that there is not a particular agent deployed at one mote, and the hardware is playing a more prominent role than in all the other elements from the Wireless Sensor Network. However, judging from the fact that this element has no other functionality that working as a gateway (from a paradigmatic point of view, carrying data from one environment to another), sink (from a data perspective, since it is going to receive all the information from the WSN) or base station (from a system point of view, as one of the elements involved in it) it does not look as necessary to upload an agent, since the required data conversions are done by using hardware rather than software.

3.1.1.2 Broker agent.

The Broker agent is probably the most important element of the WSN once it is loaded with the nSOM middleware architecture. This is due to two reasons:

- Any new service that springs up in the Wireless Sensor Network will register itself against the Broker, so it keeps all the available services registered. If any request for a service is done by the user, it will have the Broker as their first (or second, if the base station is taken into account) step in the WSN, as the Broker agent redirects the request towards the mote that is capable of providing the service. It also happens the other way around: when the answer is leaving the WSN, the Broker will direct it back to the base station as the last (or second to last) stage of the communication.

- When a composed service is requested, as it will be seen in the next part, the Broker agent will manage all the inner interchange of messages that may be needed, not requiring any further work from the human user.

Therefore, according to its functionalities it can be imagined that if the Broker agent becomes incapacitated, the whole middleware architecture will be unable to work normally, so the Broker agent must always be uploaded to the mote in best conditions (best radio signal, highest battery level, etc.).

3.1.1.3 Orchestrator agent.

Orchestrator agent is critical to correctly provide composed services; without it, they simply cannot be delivered. Usually, the Orchestrator Agent assumes the role of publishing the composed services present in the Wireless Sensor Network. So, when there is a request of a composed service, the Broker agent will re-route it to the Orchestrator agent. In fact, though, the Orchestrator does not collect any data required for a composed service, but it is aware of the simple services that are needed, so it will ask for them to the Broker agent until it has all the required data, calculates the value from them, and sends it back to the Broker.

An explanatory definition of these actions is provided in section 3.2.4, and a conceptual description for simple and composed services in sections 3.2.1 and 3.2.2.

3.1.1.4 Data providing motes (other agents).

These are the agents located on the nodes of the Wireless Sensor Networks devoted to information harvesting. Although these agents are still pieces of software, the services they represent are dependent on the data that is collected from the environment; therefore, they are closely linked to the sensors that the device they are uploaded on has mounted (unlike the Broker and Orchestrator agents, that were performing purely software functionalities regarding data request and delivery, with no mapping of any service from the “physical” world).

It is important to note that although these agents collect data from sensors, they can use actuators to offer information too –if that is compliant with the intentionality of the systems is another story-. Sun SPOT motes, for example, have an array of 8 LEDs that can be used to offer information to one human user, thus performing actuator functionalities.

All in all, they will provide readings from data belonging to the context they have been placed. In our scenario, context information regarding indoor temperature was collected (although depending on the used mote, luminosity, humidity and other information can be retrieved too), as well as data provided by an agent used to port Bluetooth-formatted information from an electronic belt, as it is developed in Chapter 4.

3.1.2 Service Discovery Message Specifications.

In order to have messaged interchanged properly, a generic PDU (Protocol Data Unit) has been defined for the nSOM middleware architecture. As it is usual, this PDU consists of two elements: a part named PCI (Protocol Control Information) as the header of the PDU, that will be providing information about the nature of the communication (the protocols being used, the type of message that is being sent –next to this part are all the possible messages that can be used- and the length of the payload) and a SDU (Service Data Unit) that will incorporate all the payload content. The payload is composed by a certain number of Objects, where an object is a field with a piece of information about the procedure (for example, if the serial number of the mote has to be included in the communication, it will be carried by an object) that has its own header (with a key value describing the object payload, and the length of the payload of the object itself) and payload (the actual relevant content of the object).

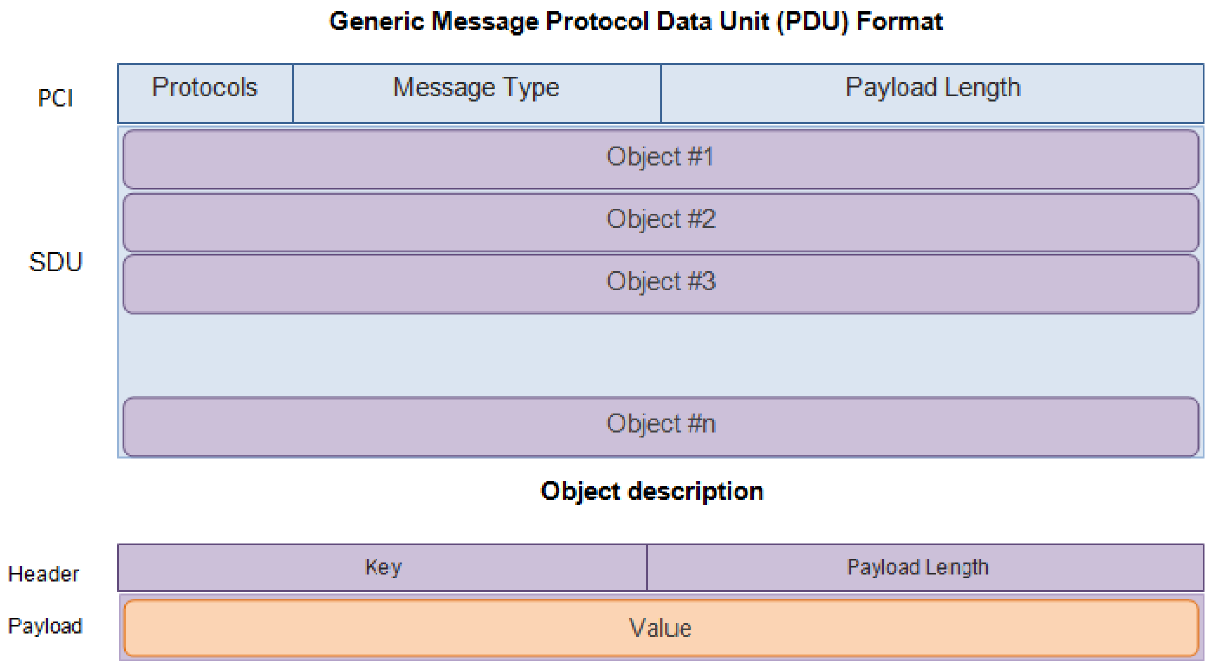


Figure 51: layout of a PDU and an object inside [Lifewear 12].

Objects can be responsible for very different tasks, as it can be learnt from the next chart:

Object acronym	Extended text
DCDO	Disconnection Code Description Object
DIO	Device Identifier Object
DMNAO	Device Model Name Object
DMNUO	Device Model Number Object
DMO	Device Manufacturer Object
DMURLO	Device Model URL Object
DNO	Device Name Object
DSNO	Device Serial Number Object
DTDO	Device Type Description Object
EDO	Event Description Object
SDO	Service Description Object
SEO	Service Event Object
SIO	Service Identifier Object
SOIO	Service Operation Invocation Object
SQO	Service Query Object
SSDO	Service Subscription Description Object

Chart 13: objects used in nSOM messaging procedures.

There are different messages that will be sent from one node to another, depending on the actions that have to be taken. Note that this messages are sent and received at the network layer; it is important to make a difference between these and the JSON requests and answers that are going to be sent at the application layer. These messages are:

1. **Hello message,** used for the advertisement of the features present in the wearable device and the description of the services that can be provided.

Source:	Wearable/WSN node
Destination:	Broker agent
Transmission mode:	Multicast
Objects required:	DMO, DTDO, DSNO, DNO, DMNAO, DMNUO, DMURLO, SDO (as many as present services)

Chart 14: main features of the Hello message.

2. **Bye message,** used to announce the disconnection of the wearable device. The payload will include a code with information about the disconnection (normal or abnormal).

Source:	Wearable/WSN node
Destination:	Broker agent
Transmission mode:	Multicast
Objects required:	DCDO

Chart 15: main features of the Bye message.

3. **Service Subscription Notification message**, used by the Orchestrator agent to subscribe to the services that are going to be announced by the other nodes.

Source:	Orchestrator agent
Destination:	Broker agent
Transmission mode:	Multicast
Objects required:	SSDO (as many as present services)

Chart 16: main features of the Service Subscription Notification message.

4. **Service Availability Notification message**, used by the Broker to notify (taking into account the subscriptions already formalized by the Orchestrator agent) that a requested service has been published by a node.

Source:	Broker agent
Destination:	Orchestrator agent
Transmission mode:	Unicast
Objects required:	SIO, SSDO (as many as present services)

Chart 17: main features of the Service Availability Notification message.

5. **Service Un-subscription Notification message**, used to cancel a previous service subscription. It is transmitted on the opposite direction than a Service Availability Notification Message.

Source:	Orchestrator agent
Destination:	Broker agent
Transmission mode:	Unicast
Objects required:	SIO (as many as present services)

Chart 18: main features of the Service Un-subscription Notification message.

6. **Service Shutdown Notification message**, used to announce that a service that the Orchestrator was subscribed to is not available, at least for the time being.

Source:	Broker agent
Destination:	Orchestrator agent
Transmission mode:	Unicast
Objects required:	SIO (as many as present services)

Chart 19: main features of the Service Shutdown Notification message.

7. **Service Registering message**, used to declare the features and services provided by a device, from the Broker agent to the mote that acts as the Gateway.

Source:	Broker agent
Destination:	Gateway node.
Transmission mode:	Unicast
Objects required:	DMO, DIO, DTDO, DSNO, DNO, DMNAO, DMNUO, DMURLO, SIO, SDO (as many as present services)

Chart 20: main features of the Service Registering message.

8. **Service Unregistering message**, used to announce a controlled service disconnection.

Source:	Broker agent
Destination:	Gateway node.
Transmission mode:	Unicast
Objects required:	DIO, SIO (as many as present services)

Chart 21: main features of the Service Unregistering message.

9. **Probe message**, used to search for a service in the ubiquitous domain. An object unused in any other message is required in this case: Service Query Object or SQO.

Source:	Broker agent
Destination:	Wearable/WSN node
Transmission mode:	Multicast
Objects required:	SQO (as many as services are searched)

Chart 22: main features of the Probe message.

10. **Probematch message**, used as an answer for a Probe message when the requested service is found.

Source:	Wearable/WSN node
Destination:	Broker agent
Transmission mode:	Unicast
Objects required:	SDO (as many as present services)

Chart 23: main features of the Probematch message.

11. **Service Data Request message**, used to request a service. This message is usually re-routed to the mote that is offering the service.

Source:	Gateway, Broker, Orchestrator
Destination:	Broker, Orchestrator, Wearable/WSN node (respectively).
Transmission mode:	Unicast
Objects required:	SDO, SOIO

Chart 24: main features of the Service Data Request message

12. **Service Data Response message**, used as an answer to the previous message. Service Event Objects are used here; each of them will contain data about the requested service.

Source:	Wearable/WSN node, Orchestrator, Broker
Destination:	Orchestrator, Broker, Gateway (respectively)
Transmission mode:	Unicast
Objects required:	SDO, SOIO, SEO

Chart 25: main features of the Service Data Response message.

3.2 Functionalities of nSOM.

Conceptually, there are two main functionalities that nSOM provides: simple and composed services. While simple services are quite self-explanatory, composed services require to define what sensor virtualization is about, in order to have a more accurate knowledge of the whole concept.

3.2.1 Simple services.

A simple service is a service that is offered by an existing agent, usually uploaded in a Wireless Sensor Network node. It is related with a piece of information provided by a sensor, and will just deliver it when the agent is requested to do so. The information provided is not stored in any intermediate component, but it will be delivered to the agent that requested it in the first place; it will be a process where the sensor will try a reading at the environment and will take the result to the operating system (or in our case, the mini Java Virtual Machine Squawk) which is the entity that will send it to the middleware as the answer that is expecting the application layer, in a fashion resembling the model that appeared in the figure number one of this Final Degree Dissertation.

Examples of simple services all the data that a sensor mounted on a mote can provide that have been mentioned before: temperature, humidity, etc.

3.2.2 Composed services and sensor virtualization.

A unique feature of the nSOM middleware architecture is that it is able to offer one single service based on different readings collected by the Wireless Sensor Network. In this case, the service will not get provided as a result of a request to an agent that, in the end, is consulting a sensor, but as the result of a data processing action, where the data to be processed has been collected by doing several inner requests for different simple data, without further user intervention. These inner requests are necessary because the response that is going to be delivered is dependent on the values of the readings that have been gathered as the data from the simple inner requests, and the service that is attended like this is a composed service

For the final user, the service is provided as if it was a simple service with no difference with the others. Furthermore, when the Broker agent directs the request for this service to the Orchestrator agent, it is done like this because the Orchestrator agent has published the composed services as if they were simple services that would be attended by “listening” a sensor reading. Thus, for the final human user and even for part of the Wireless Sensor Network, the request will be attended the same way as if there was a “virtual” sensor to deliver the data that will solve the request, although in fact there is none like that and the response is offered by a different way. This is how sensor virtualization can be defined.

It can be understood by having a look at the next figure. At the left side of it the answer for a simple service request is delivered in a predictable way; at the right side, the answer for a composed service is done the same way for the user, but it is obtained by processing data from two different stacks (and therefore, two different sensors and two different devices):

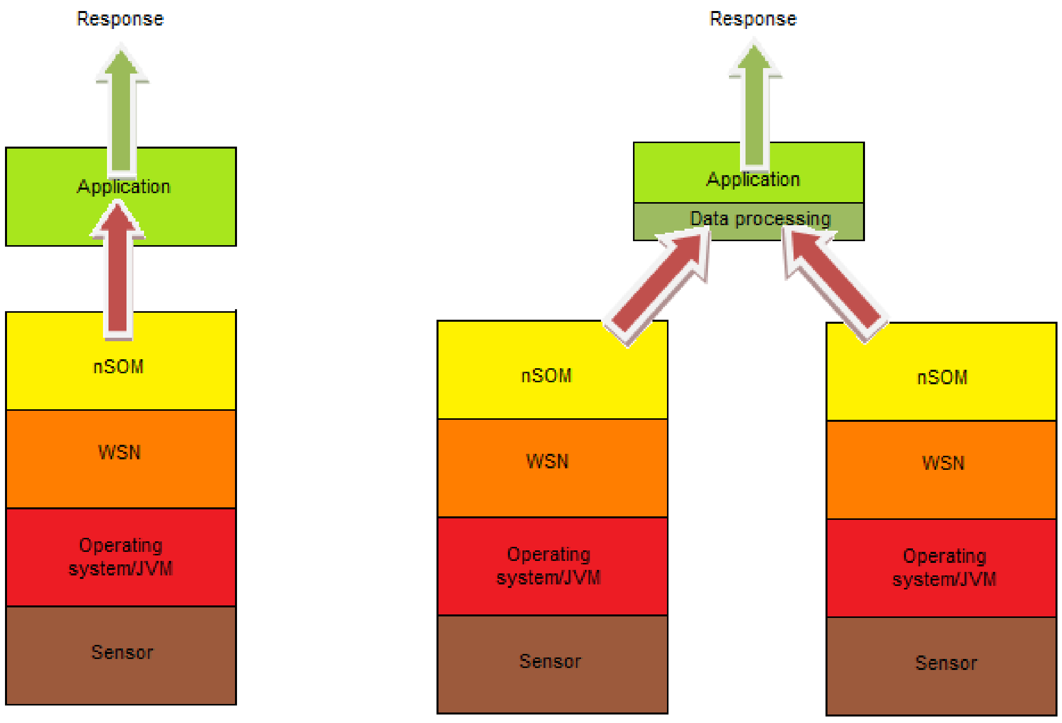


Figure 52: comparison between a simple and a composed service.

All the communications that have been explained before are going to be cleared up by using sequence diagrams in the next two sections of the Final Degree Dissertation.

3.2.3 Simple services communication model.

The sequence diagram that describes a simple service communication model for nSOM middleware is as follows:

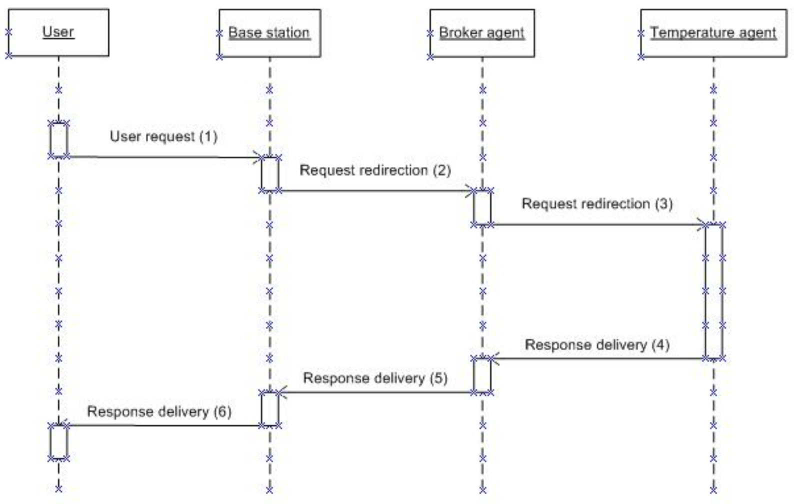


Figure 53: sequence diagram for a simple service.

In the sequence diagram a temperature request (which actually can be invoked in our real scenario) is used as an example for how simple services work under semantic middleware architecture nSOM: when the human user requests a temperature, the request is introduced by using a keyboard, a touch screen or whatever device is available at the moment (in our system an ESB has been mounted on a PC to guarantee that requests from almost every kind of device can be attended, as long as the device is able to route a request from a IP version 4 address) and is redirected via a conventional network (wireless or not, depending on the device from it was made) to the base station **(1)**. After that, as long as the requested service is available in the system, the base station will always redirect it to the Broker agent, the first step on the communication **(2)**, as it is the entity of the WSN that is aware of what services are functional. If the service is recognized by the Broker agent (that will be uploaded on a node of the WSN, most likely a mote), the request will be redirected again towards the node that has installed the agent (in this particular case, the temperature agent) capable of taking the data from the sensor present at the piece of hardware **(3)**.

Once the information has been obtained, it has to be delivered to the human user. To accomplish this, the service response will undo the route done before by the request. To begin with, the answer will be carried to the Broker agent, since all the communications are done by following a hierarchy where the Broker will receive all the responses to the requests **(4)**. Once the Broker has received the value, it will be transmitted back to the base station **(5)**, and the base station, bridging the Wireless Sensor network and the non-ubiquitous network, will finally send it to the device where the final user started the request in the first place **(6)**.

The sequence shown here is located at the application layer; the vehicle to transport the requests and answers are JSON messages. JSON is more useful than XML under this ubiquitous environment because, although XML supports standardizing schemas, it is usually more verbose than JSON, consuming more bandwidth and energy resources.

Should any of the stages formulated before fail, the request will be unattended; an answer mechanism must be implemented in order to notify the user about the request failure (in our deployed system it was done so, as it will be learnt in the next section) and let them perform the same or other petition again.

3.2.4 Composed services communication model.

The sequence diagram that describes a composed service communication model for nSOM semantic middleware is as follows:

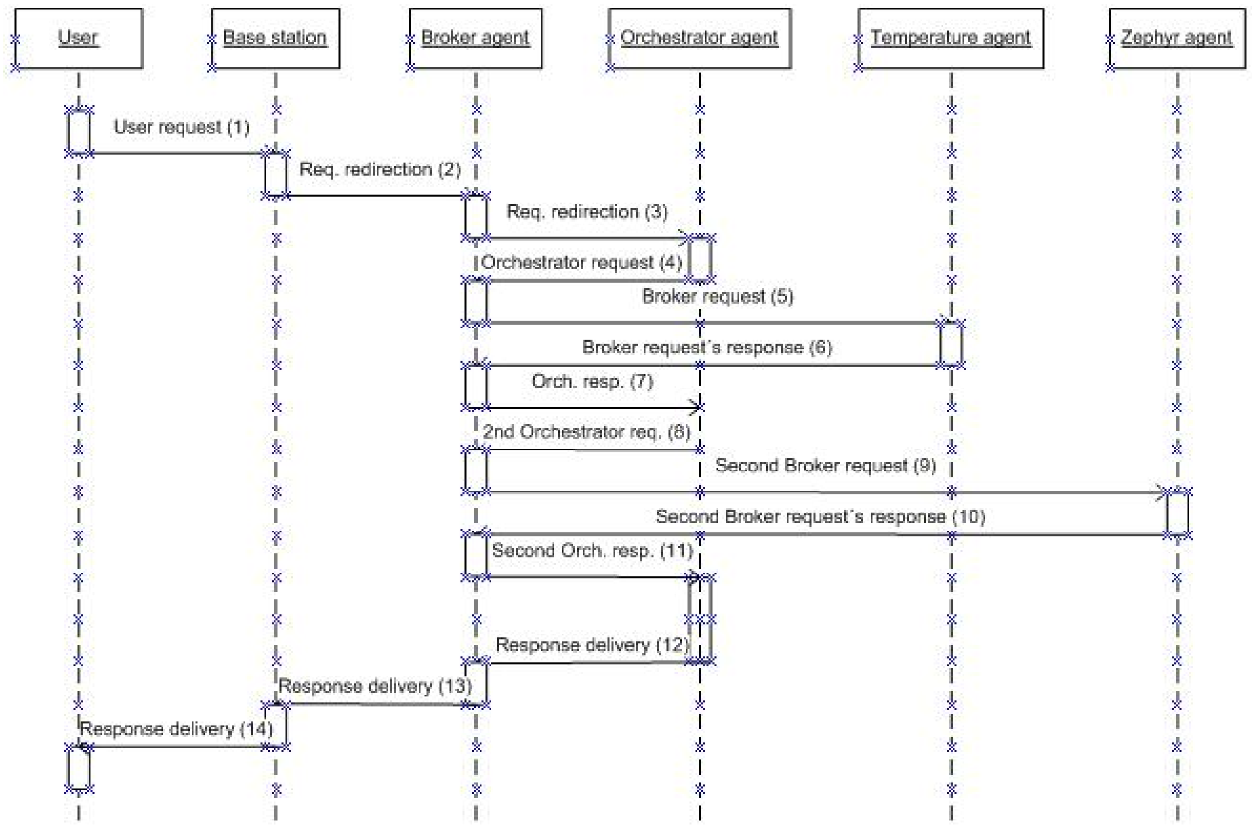


Figure 54: sequence diagram for a composed service.

To use as an example of a communication model for a composed service, a real service that can be invoked from our deployment -Temperature Control- is explained here. Temperature control is a composed service that will evaluate the values from the environmental temperature (extracted from a mote with a Temperature agent deployed) and the body temperature (extracted from the mote with the agent that is porting all the data obtained via Bluetooth from a Zephyr-marketed belt). The result of the evaluation of temperatures (very high, high, medium, low or very low) is what will be sent to the user. Note that in this case this evaluation of the two separately received temperatures is the data processing stage, for other composed services data processing might represent other actions.

As it was done in the previous case, the followed steps are explained here: the user will request the service **(1)** as if it was a simple service -and therefore there is no difference for the user; it will be invoked the same way, otherwise there would be no real sensor virtualization-. The request will reach the Broker agent, the element of the WSN that has received the service registration from all the other agents **(2)**. As this is a composed service the request will be redirected to the Orchestrator agent, as it is done with all the composed services **(3)**. The Orchestrator agent is incapable of providing a value for that service because it cannot be attended the way that was done before; despite the impression given to the final user, there is no sensor measuring the level of Temperature Control. Nevertheless, the Orchestrator agent is aware that once the values of the environmental and body temperature come to its grasp, they can be processed to have a satisfactory response. So, the Orchestrator agent will ask the only entity of the network that is aware of all the deployed agents -that is, the Broker agent- for the two values that it requires, beginning with the environmental temperature **(4)**.

In this way, the Broker agent will send a request to the Temperature agent deployed in one mote **(5)**, asking for the value of the environmental temperature. Once the value is send by the Temperature agent to the Broker agent in a JSON message **(6)**, it will re-route the answer to the Orchestrator agent **(7)**; note that at this point the Broker agent ignores the procedure that is taking place at the Orchestrator agent, nor it is expecting another request from the latter agent. However, a single value is not enough to determine the result of the request made by the user (nor it would be sensible to have all this procedure for a single value: it could have been retrieved as a simple service); the body temperature is needed too, so the Orchestrator agent will request it to the Broker agent **(8)**, expecting to be answered as it was done before. And in fact, the request is attended the same way: the Broker agent will ask for the data to the agent that it knows that can provide it, which happens to be the Zephyr agent **(9)**, and this agent will offer it if there is no unforeseen trouble **(10)**. As soon as the Broker agent receives the JSON response with the particular datum, it will send the whole JSON message to the Orchestrator **(11)**, the entity that requested it and is capable of isolating it from all the other content of the JSON message received.

Now that the Orchestrator agent has the two values, it has all the information required to do the evaluation. Once it has taken place, one of the five possible results commented before will be placed into another JSON response message and sent back to the Broker agent **(12)**. Able to distinguish a request from a response, and having as response destination the same node all the times, this Broker agent will send the response to the base station **(13)**, and finally it will be re-routed to the device where the final user started all the process **(14)**. If there is any failure while sending the messages or retrieving the data, two solutions can be tried: either the final user is notified about it, and they are offered the chance to request the service again, or if one or several of the required values to obtain a result for a composed service request has already been retrieved, the needed processing is undertaken without bearing in mind all the other required values.

Chapter 4

System validation and results exposition

4.1 Justification of the scenario.

If nSOM is desired to be part of a ubiquitous, functional subsystem, it must be installed under very specific conditions. What follows now is a complete description of a system that was deployed as a tangible reality in a real scenario. The system had the features that have become typical of ubiquitous, pervasive computing: low-capability devices to get context information, deployment of the nodes at places were conventional equipments would have it hard to be placed and seamless integration of all the components on the environment where they are located.

What is more, the system was deployed with the idea of providing data not only about the context of the system, but also from a final user that is wearing a belt capable of retrieving body information in real time (body temperature, heart rate, breathing rate). This information can be accessed by requesting it as a simple service, but with the wide range of available data, composed services can be invoked as well, looking to have for the whole system an added value that allows it to make a difference from other ubiquitous middleware architectures.

All the hardware devices presented, and how the software is mounted on them are issues that will be explained in the next sections.

4.1.1 User cases of the scenario.

According to the analysis that was made before implementing nSOM, and before a real system had this architecture installed, user cases were studied. The result obtained was like this:

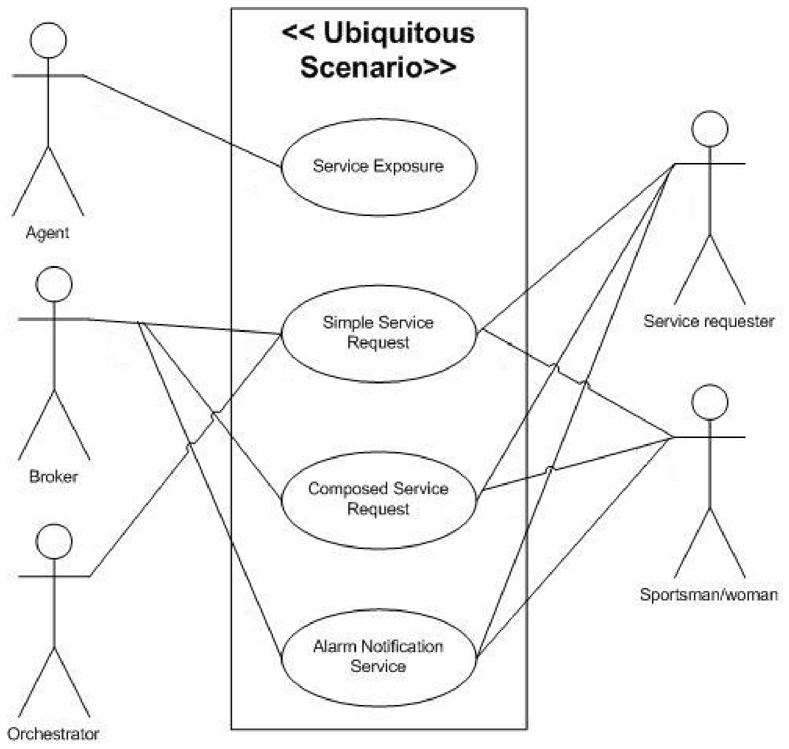


Figure 55: user cases of the scenario.

The actors involved in this ubiquitous scenario, along with the different user cases presented, are going be extended for them to be completely understood.

4.1.1.1 System actors.

As it has been established, there are five actors that are part of this system. Those actors are:

- **Agents**. These entities will be granting the data (from the context or from the user) that will be presented when simple or composed services are invoked. For this to happen they must publish themselves on the Broker agent, though.
- **Broker**. It will be responsible for the already mentioned functionalities typical of them, being part of simple and composed service requests and responses.
- **Orchestrator**. Its main duties have not changed; it will be retrieving the simple data needed to fulfil the composed service requests.
- **Service requester**. They will usually be human beings either requesting simple or composed services from the system. They may also receive alarms, depending on their activities.
- **Sportsman/woman**. While they can also request for services, they are more likely to receive alarm warnings. It is supposed that service requesters and sportsmen/women are at least two different users (one requesting services and the other focused on their own activities), but the system could be used for just one person too.

4.1.1.2 Different user cases.

The different user cases that were presented in the previous figure will be exposed here:

4.1.1.2.1 Service exposure.

This service will be used just for the agents to subscribe at the Broker agent, and therefore publish their services. Although a rather “passive” service, with no human user or other nodes intervention, it is pivotal to offer the services, since only those registered at the Broker agent will be able to be invoked. If a service has not been registered and it is requested by a user, the request will be dismissed.

Requesting entity	None, the agent will perform it whenever the mote it is deployed on is reset.
Result	The agent and its services are registered by the Broker agent.
Required data	None, a functional agent.
Other considerations	None.

Chart 26: features of the Service exposure user case.

4.1.1.2.2 Simple service request: temperature request.

This service is used whenever the temperature of the context where the system is located is wanted. As a simple service it can be requested by a human and the Orchestrator agent too, in case it has been required for a composed service.

Requesting entity	Service requester, Sportsman/woman (rarely), Orchestrator agent.
Result	The environmental temperature is provided. If the request fails, -100º is offered as an answer (not included in a JSON message).
Required data	None, a functional Temperature agent.
Other considerations	None.

Chart 27: features of the temperature request user case.

4.1.1.2.3 Simple service request: body temperature request.

This service is used when the body temperature is requested, either by a human user or by the Orchestrator agent, in order to check if there is any alarm going on. This information is collected from the Zephyr belt.

Requesting entity	Service requester, Sportsman/woman (rarely), Orchestrator agent.
Result	The body temperature is provided. If the request fails, -100º is offered as an answer (not included in a JSON message).
Required data	None, a functional Zephyr agent and a Zephyr belt.
Other considerations	It requires an element (the Zephyr belt) external to the Wireless Sensor Network. When the answer is received, the value and a four digit belt identifier are provided altogether.

Chart 28: features of the body temperature request user case.

4.1.1.2.4 Simple service request: heart rate request.

This service differs very little from the previous one; this data will be requested by a human or by the Orchestrator agent, and will be provided by the mote receiving the data from the Zephyr belt.

Requesting entity	Service requester, Sportsman/woman (rarely), Orchestrator agent.
Result	The heart rate is provided. If the request fails, the value -100 is offered as an answer (not included in a JSON message).
Required data	None, a functional Zephyr agent and a Zephyr belt.

Other considerations	It requires an element (the Zephyr belt) external to the Wireless Sensor Network. When the answer is received, the value and a four digit belt identifier are provided altogether.

Chart 29: features of the heart rate request user case.

4.1.1.2.5 Simple service request: breathing rate request.

Again, this service is provided by the mote connected via Bluetooth to the Zephyr device.

Requesting entity	Service requester, Sportsman/woman (rarely), Orchestrator agent.
Result	The breathing rate is provided.
Required data	None, a functional Zephyr agent and a Zephyr belt.
Other considerations	It requires an element (the Zephyr belt) external to the Wireless Sensor Network. When the answer is received, the value and a four digit belt identifier are provided altogether.

Chart 30: features of the breathing rate request user case.

4.1.1.2.6 Composed service request: injury prevention request.

This, as the next composed service, can only be invoked from a human user. Unlike all the others, this is more likely to be asked by a sportsman/woman so they can check if they are taking their physical exercises to dangerous levels.

In order to define the three levels of risk that can be obtained as an answer (see other considerations paragraph), upper and lower thresholds were fixed in our scenario. For body temperature were 38.0º and 36.4º (the Zephyr device was not absolutely accurate when measuring this data and less tight thresholds had to be used), for environmental temperature 34.0º and 12.0º degrees and for heart rate 120 and 50 beatings per minute. Margins were fixed at 5º for environmental temperature, 5 heart beatings for heart rate, and 0.4º for body temperature.

Requesting entity	Service requester, Sportsman/woman.
Result	One of the three levels of risk of suffering a muscular injury by over-exercising: High risk, Medium risk, Low risk.
Required data	Environmental temperature, body temperature and heart rate.
Other considerations	When the three required pieces of information are obtained, they are evaluated. If there is at least one of them beyond or below the maximum or minimum thresholds fixed, a High risk value is sent (and it is very likely that, although independently, an alarm will be sent to the ESB and the alarms mote). If at least one of the values is within the allowed range, but inside a margin close enough to the upper or lower threshold, a Medium risk

	message is sent. Otherwise, a Low risk message is sent.

Chart 31: features of the injury prevention request user case.

4.1.1.2.7 Composed service request: temperature control request.

This service works in a manner resembling the previous one, although it requires one less piece of information (heart rate is not taken into account for temperature control purposes). The thresholds and margins fixed for injury prevention service were kept for this one.

Requesting entity	Service requester, Sportsman/woman.
Result	One of the five levels of control: Very low, Low, Medium, High and Very high.
Required data	Environmental temperature, body temperature.
Other considerations	When the two required pieces of information are obtained, they are evaluated. If there is at least one of them beyond the maximum thresholds fixed, a Very high value is sent (and it is very likely that, although independently, an alarm will be sent to the ESB and the alarms mote). If at least one of the values is within the allowed range, but inside a margin close enough to the upper threshold, a High message is sent. On the contrary, if at least one of the values is within the allowed range, but inside a margin close enough to the lower threshold, a Low message is sent. Finally, if there is at least one of the values below the minimum thresholds fixed a Very low value is sent (and again, it is very likely that an alarm will be sent). Otherwise, a Medium message will be sent.

Chart 32: features of the temperature control request user case.

4.1.1.2.8 Composed service request: alarms request.

As the other composed services, this one can only be requested from one final user. This service will notify a user, usually different from the sportsman/woman, that a value obtained from the registered nodes has been beyond the upper or lower threshold values fixed. The value will be visualized when it is requested through the ESB, while an alarm will be sent to the Alarms mote by the next service.

Requesting entity	Service requester, Sportsman/woman (rarely).
Result	Either a message noting that there are no alarms, or a number indicating the nature of the alarm and the value that has triggered it.
Required data	Environmental temperature, body temperature, heart rate.
Other considerations	The JSON message received will contain a three-figured number, where the first digit gives away the nature of the alarm (1=too low environmental temperature, 2= too high environmental temperature, 3= too low heart rate,

	4= too high heart rate, 5= too low body temperature, 6= too high body temperature) and the other two tell the value that made the alarm spring up. For example, 235 would be a high environmental temperature alarm (first digit=2), because there are 35º at the room where the sportsman/woman is performing their workout. For heart rate the whole figure is added (for example, 338 would be an alarm value claiming that the sportsman/woman´s heart is beating at 38 beatings per minute, and 475 would indicate that the sportsman/woman´s heart beats at 175 beatings per minute).

Chart 33: features of the temperature alarms request user case.

4.1.1.2.9 Alarm notification service.

This is a composed service unique and separated from the others in the fact that it is not requested, but comes up whenever one of the values that are requested (they are the same than those in the previous service) is above or below the thresholds. The alarmed value will be sent to the human user wearing the watch used in our scenario to notify alarms.

Requesting entity	Orchestrator Alarms agent; it always reaches the Sportsman/woman.
Result	A beeping sound on the Android watch, and a warning on its screen notifying that a threshold has been surpassed.
Required data	Environmental temperature, body temperature, heart rate.
Other considerations	There is one single agent devoted to provisioning this service (Orchestrator Alarms agent). This service is not invoked by a human user.

Chart 34: features of the alarm notification user case.

4.2 Description of the validation scenario.

Following the previous points of the Final Degree Dissertation, along with all the services that are available for human users, a figure with all the elements present in the scenario is offered at the start of the next page [GRyS&12]:

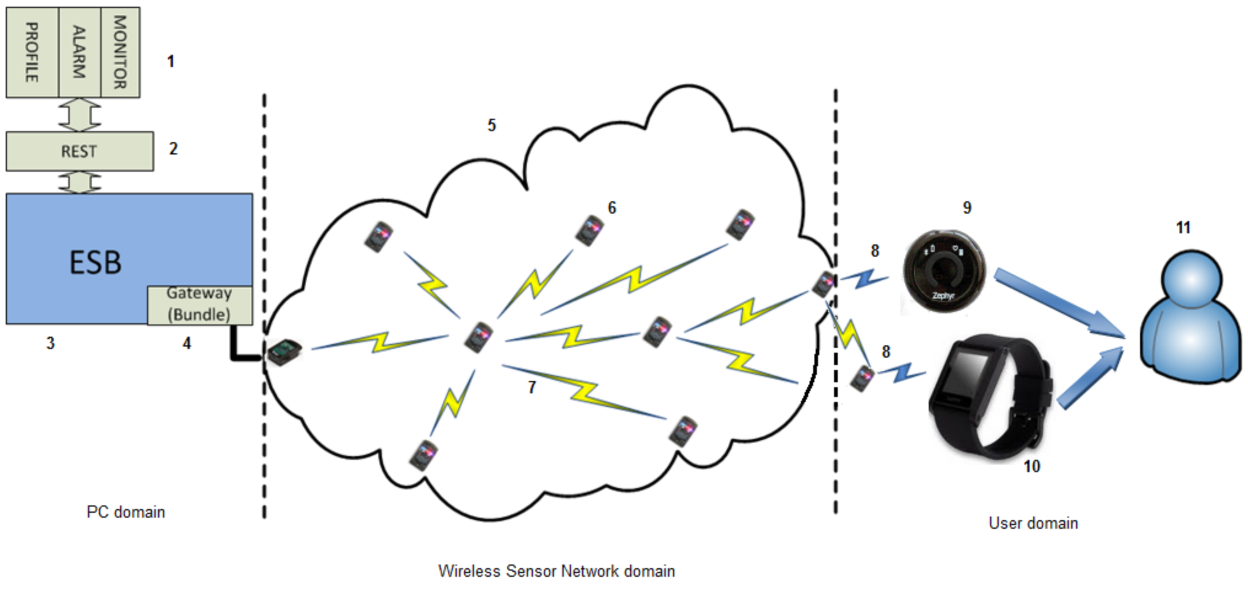


Figure 56: a holistic view of the system.

In order to describe them all, and taking the previous figure as a starting point, hardware and software points of view are used to have a wider approach on the elements present in the system.

4.2.1 Hardware elements of the system.

The physical entities present in the system are as follows:

- A Personal Computer with an Ubuntu distribution as the operating system (**PC domain**). Ubuntu will have an important role in the PC domain, as it is the operating system chosen to install the ESB component that is going to be receiving all the requests from devices belonging to end users. In addition to the ESB (3), the PC will also be mounting the software bundle required to have the base station processing the requests (4) and a REST interface (2) as a gateway between the proper ESB and the elements present in a mobile device.

- A mobile device (or more likely in our particular case, a **mobile phone**), used to store all the information associated to a particular user, such as the profile (1) -height, weight, gender, etc.-. Along with the profile, if an alarm springs up it will be sent from the Wireless Sensor Network (or the mote with the Zephyr agent deployed, depending on where it was triggered) to the application storing the profile information, which also happens to monitor the user by requesting the heart rate every two seconds, along with the requests performed to the national Spanish meteorological database (AEMET, Agencia Estatal de METeorología). The software installed in the mobile phone was programmed by SAI Wireless.

- A **Wireless Sensor Network** (5) that will be behaving as an orthodox ubiquitous system. The WSN has motes scattered in an environment (according to the objectives of UPM in the Lifewear project it is an indoor wide room, that is, a gymnasium) measuring three different temperature values (6). Sun SPOT motes where used as the hardware of choice for the **WSN nodes** due to their RAM and ROM capabilities and their low energy consumption. These motes will communicate to each other by using the **standard 802.15.4** (7), which is specifying the physical layer and the MAC (Medium Access Control) layers for Personal Area Netowrks [Radio-electronics 12] or, in this case, Wireless Sensor Networks. Note that 802.15.4 is not Zigbee; Zigbee is an industrial alliance scoping only the top of the 802.15.4 stack [Sun SPOT 12b], not the whole standard.

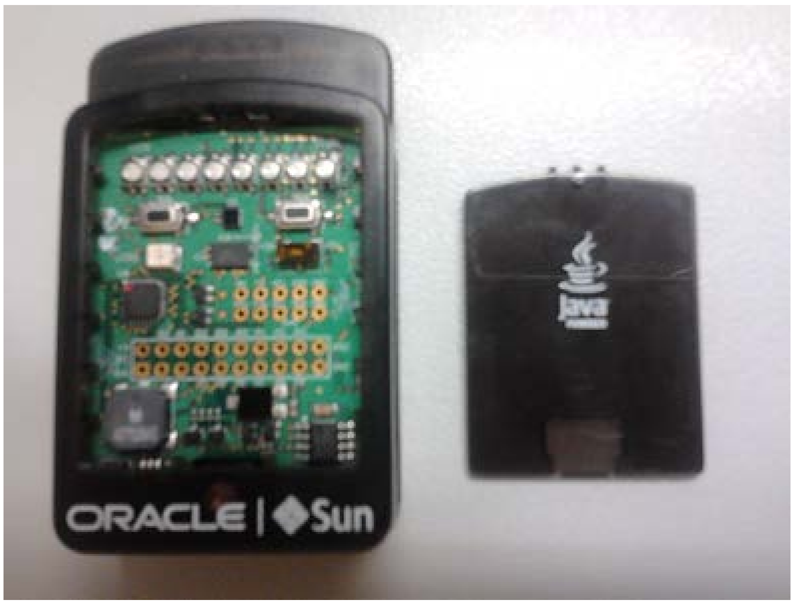


Figure 57: one of the actual motes used in the deployment of the system.

- At the user domain, the **human user** (11) will be carrying several devices on. Firstly, two motes, one with the Zephyr agent that is porting the Bluetooth data from the Zephyr belt, and another one that will receive an alarm notification, should there be any value out of the range fixed for the system by the thresholds (note that while the mote with the Zephyr agent deployed can be considered as an endpoint of the Wireless Sensor Network, the other mote will not communicate with the WSN at all and will just receive an alarm notification, without sending any piece of information to the WSN. Nevertheless, this mote and the former are communicating via **Bluetooth data converting boards** (8) with the two other user devices: a **Zephyr BioHarness™ v3 belt** (9) and a **WIMM Android programmable watch** (10). The Zephyr belt looks as follows:

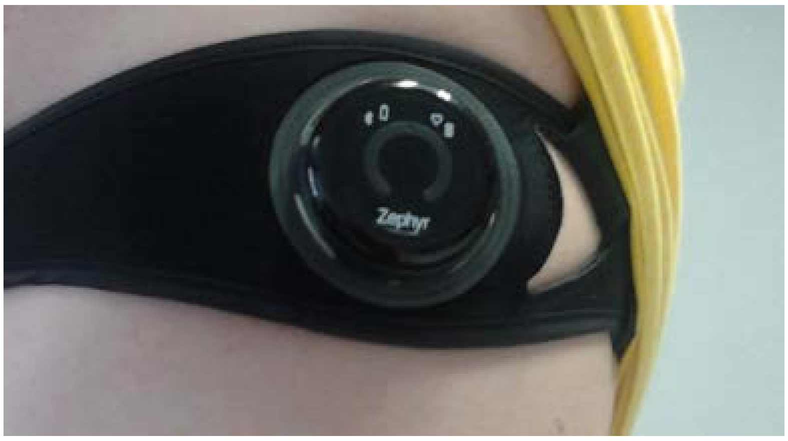


Figure 58: Zephyr BioHarness™ v3, worn by a final user.

According to its data specifications [BioHarness™ 3 10], Zephyr BioHarness™ v3 is a belt capable of measuring different types of human body data (body temperature, breathing rate, positions and postures, etc) and given that it was needed a device that could be worn by a person while doing sport or performing a workout, it suits fine for our purposes. This belt is the device collecting the body-related parameters used for our system (body temperature, heart rate, breathing rate) but since the data are transmitted via Bluetooth and Sun SPOT motes do not support it natively, their hardware had to be augmented by using Bluetooth data converting electronic boards. An electronic board model, suitable enough due to its size and its capabilities, is marketed by Sparkfun Electronics –model Bluegiga WT-32-, so two Bluegiga boards were purchased [Bluegiga 12]. One board was attached to the mote that was charged with the task of converting the Bluetooth required parameters into data that could be transferred throughout the Wireless Sensor Network; the other to the mote that would communicate with the WIMM programmable watch.

All this work was done primarily by Alexandra Cuerva with the assistance of other people. For more information on how the Sun SPOT mote´s pins were connected to the Bluetooth boards, and how the Bluetooth connection was made possible, it is strongly encouraged to have a look at Alexandra Cuerva´s own Final Degree Dissertation [Cuerva 12].

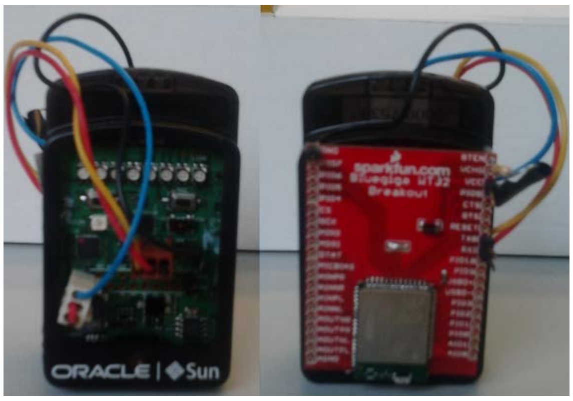


Figure 59: frontal and rear view of a mote augmented with a Bluegiga WT-32 board.

Finally, another device was required to notify the user of any alarm that would come up regarding their physical conditions (too high heart rate, too low body temperature, etc.). To accomplish this task, another device was purchased: a programmable Android watch from a vendor named WIMM (the watch has been named WIMM One) [WIMM 12]. What was interesting for our deployment is that this watch could be programmed to have events notified: if, for example, there was an alarm regarding a too high heart rate, the watch could be programmed to display it at its LCD screen, along with a beeping sound to warn the final user. Again, the programming effort was done by Alexandra Cuerva [Cuerva 12].

The next figure is showing all the physical devices that were used for the system that was mounted for this Final Degree Dissertation, the one where data regarding the performance of the network was collected from. All in all, seven motes were used (three to deploy three different Temperature agents, one to deploy the Broker agent, another one to deploy the Orchestrator agent, and two more for the Bluetooth data conversion and alarm notification duties), the Zephyr BioHarness™ v3 belt and the WIMM Android watch (note the Lifewear logo on the watch screen). For ITEA2 evaluation purposes, another wider setup was built on 11th June 2012; this architecture was deployed in the gymnasium facilities of the EUITT, and among other equipment, there was another smaller wireless Sensor Network deployed by Tecnalia [Tecnalia 12], composed by two Sun SPOT motes and a base station connected to UPM´s ESB. There were some issues related with the coexistence of both WSNs; it was likely that there were interference phenomena between them that crippled the communications among nodes. These issues were solved by giving unicast communications a more prominent paper and, after thorough tests, changing the frequency at which data was requested for the alarm service.

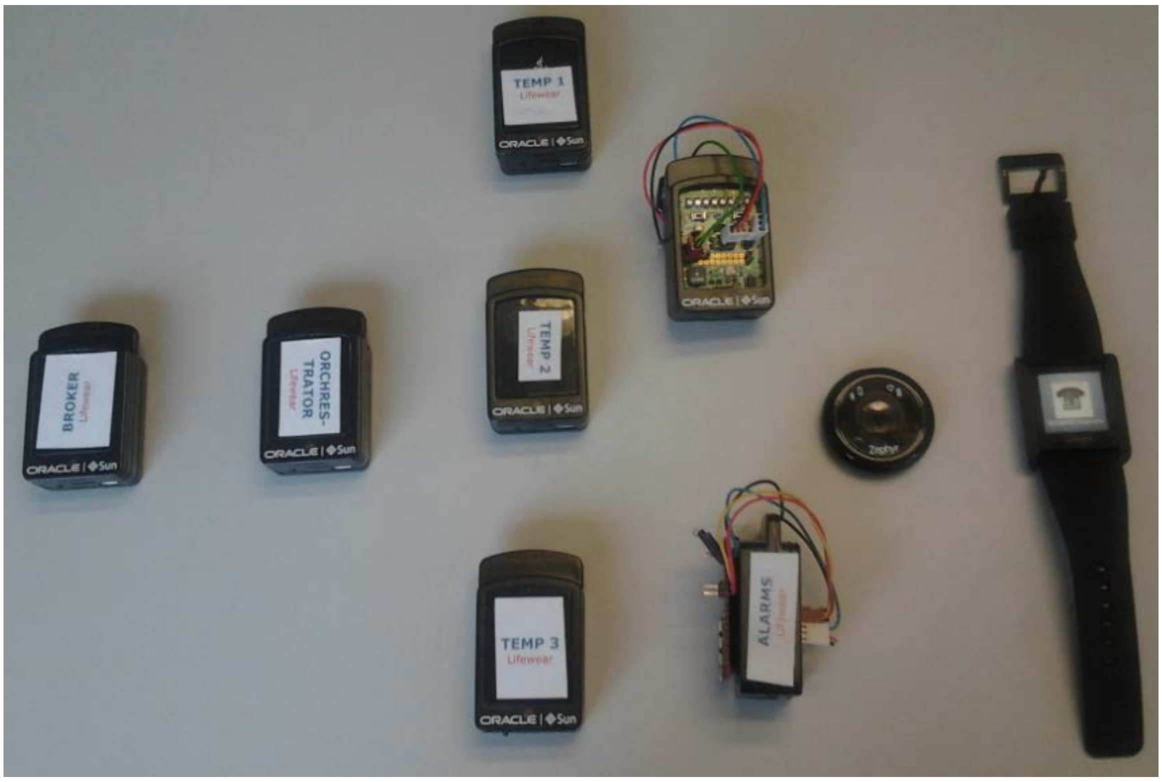
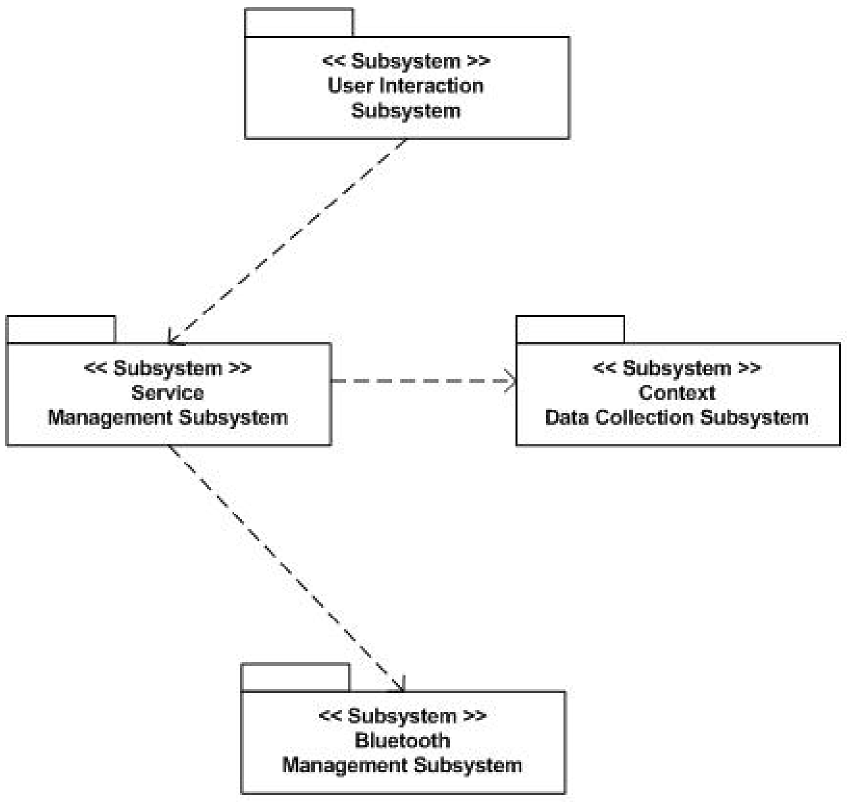


Figure 60: devices used for the data-collecting scenario.

4.2.2 Software elements of the system.

There were as many software elements present on the system as hardware-related ones. Most of them were agents with purposes related strictly to the Wireless Sensor Network; others were out of the Wireless Sensor Network but inside the system nevertheless.

From a software perspective, our architecture can be separated into four different subsystems, each one with a different concern. User Interaction subsystem will be focused on all the duties related to the successful retrieval of requests from the user. Service Management subsystem will be bent on taking the necessary actions to obtain the requested information from the Wireless Sensor Network. Context Data Collection subsystem will collect the information that is related with the context where the system is deployed. Finally, the Bluetooth Management subsystem will be taking care of all the issues related with data collection from the Zephyr belt, and alarm delivery on the mote that is connected via Bluetooth to the Android watch. The subsystems are relating each other in a particular way: the User Interaction subsystem will ask for the information to the Service management subsystem, which will have gathered it from requests done to Context data Collection and Bluetooth Management subsystems.



<< Subsystem >> User Interaction Subsystem
<< Subsystem >> Service Management Subsystem
<< Subsystem >> Context Data Collection Subsystem
<< Subsystem >> Bluetooth Management Subsystem

Figure 61: Subsystems diagram of our system.

4.2.2.1 User Interaction subsystem.

There are two components present in this subsystem, the ESB and the User Interface.

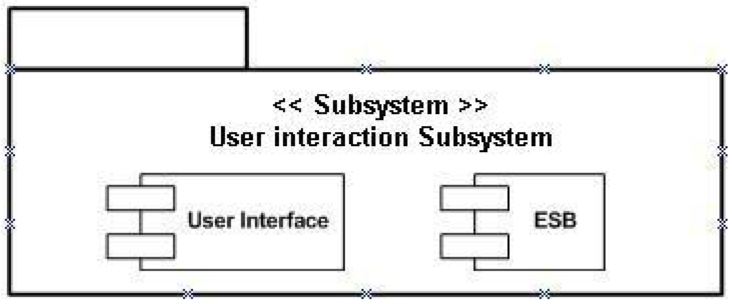


Figure 62: User Interaction subsystem and its inner components.

ESB (numbered as 3 in figure 55) is an acronym for Enterprise Service Bus. It is a software architecture model under the principles of Service Oriented Architecture that allows the integration of different technologies used by separated service invocators. In a way, it plays a role resembling that of the middleware: it will interpret the requests that it receives and, as long as they are in a format understandable by the ESB, service requesters will not have to worry about delivering their petition in a particular format, since they will be interpreted by the ESB. Once the ESB receives the request, it will resend it to one of the interfaces it has to communicate with an internal system.

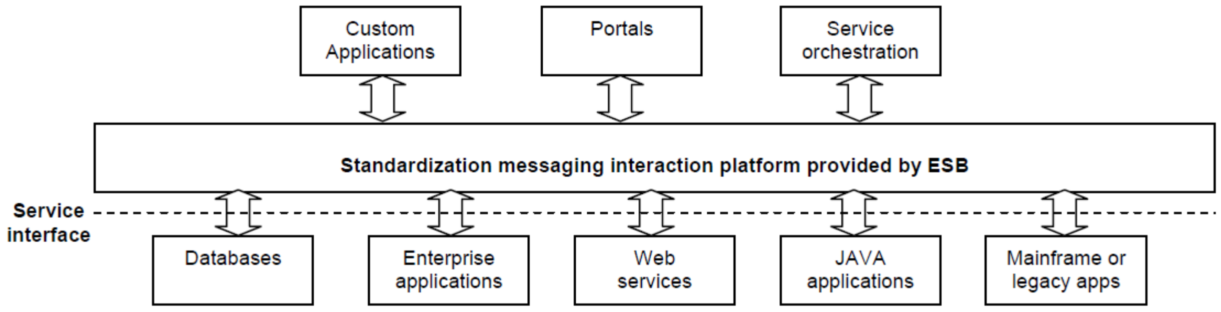


Figure 63: an ESB architecture, as in [Teo&10].

In our system, the ESB will be useful to homogenize the nature of the requests done by the final user: it has been tested with petitions originating from tablets, mobile phones and the PC where it is installed. This was the result of the work done by Pedro Castillejo Parrilla and Huang Yuanjiang.

The User interface, on the other hand, refers to the way the user requests access the ESB. It is done by two ways: on one hand, the services can be accessed via URL, where all the information related to the required IP address, the port number and the service that is going to be consulted is included. For example, if the Injury prevention service is invoked, the URL would be: **192.168.0.199:8181/cxf/crm/lifewear/injury**. This URL was using a private IP direction because it was for testing purposes; when the Lifewear scenario was deployed, or external tests had to be undertaken, a public IP address was provided.

On the other hand, the SME called SAI Wireless [SAI 12] devised an interface to access all the services they were interested in as part of their own developments in the Lifewear project. The interface was thought to be offered as support for a sports routine, and would advise the user to perform warming ups, different workouts, etc.

4.2.2.2 Service Management subsystem.

This subsystem has two components as well:

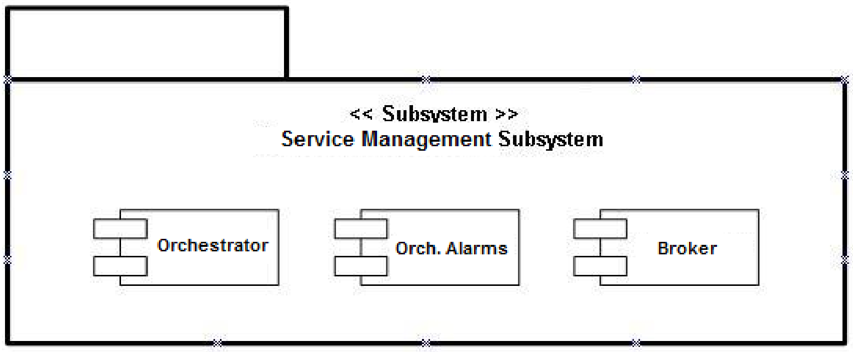


Figure 64: Service Management subsystem and its inner components.

Their functionalities of these components are what can be expected from them: the Orchestrator agent is the key component in composed services processing, requesting the Broker agent for the simple services that will be asked for, and the Broker agent is critical for the whole nSOM middleware architecture and the Wireless Sensor Network once nSOM is deployed, since it has registered which services have been announced and all the requests and responses of the Wireless Sensor network must go through it, at one point or another.

In addition to these already known agents, another one appears: the Orchestrator Alarms agent is the one that will be requesting all the needed data to check whether a value is “alarmed” (that is, above or below the values fixed for the upper and lower thresholds).

4.2.2.3 Context Data Collection subsystem.

As it can be guessed, this subsystem is responsible for measuring and providing with the context information that is proper of the environment the WSN is deployed in. For our scenario it has three different, although similarly working, components:

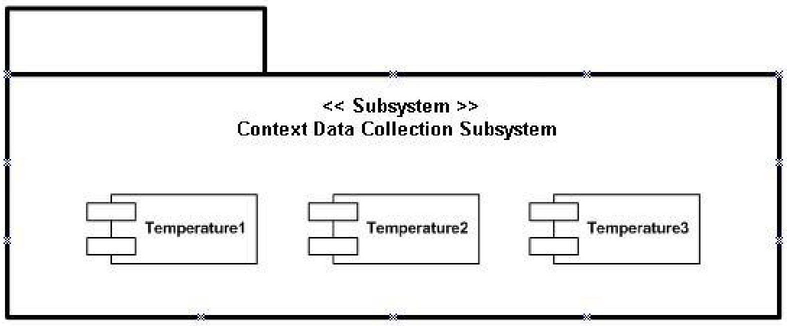


Figure 65: Context Data Collection subsystem and its inner components.

As it is observed, there are three temperature agents that will physically be deployed in three motes. The reason for having three different agents is measuring the temperature in three different places of the environment. Plus, in this case, when time measures were taken, each of the equally functional agents was under different conditions, so it is a good way to test, for example, what sort of disturbance is the worst for the network.

4.2.2.4 Bluetooth Management subsystem.

This last subsystem has two more components, as it can be seen in the next figure:

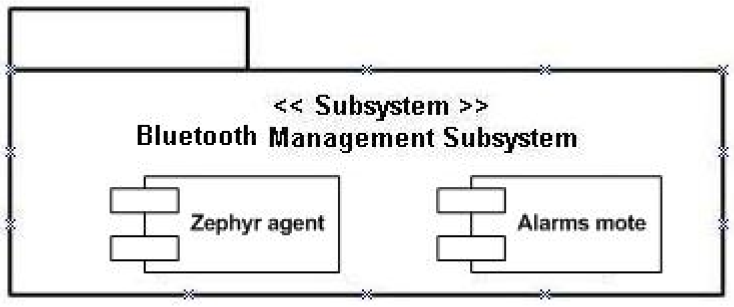


Figure 66: Bluetooth Management subsystem and its inner components.

The two components are going to deal with their already known functionalities: the Zephyr agent will adapt the data obtained from the Zephyr belt via Bluetooth connection to a format that can be understood by the other motes, and the Alarms mote will be taking the alarm information to the Android programmable watch via Bluetooth communication as well.

4.2.3 Communication on the application layer of the system.

The communication is going to be tackled by using JSON (JavaScript Object Notation) messages. Apart from the particular format required by JSON, an additional one has been establish in order to have a standardization of the communication that covers the field the WSN will be operating, and in this way the particular data can be recovered in an easier way.

The request message that will be sent from the base station throughout the Wireless Sensor Network will be formatted like this:

```
{
      "transport": "j2me.radiogram",
      "envelope": "JSON-2.0",
      "target": "<IP or MAC address>/<name of the destination agent ",
      "origin": "<IP or MAC address>/<name of the source agent>",
      {
            "operation": "<operation name>",
            "parameters": [ < parameters or void, if there are none> ]
      }
}
```

All the changing fields have been highlighted with bold characters; as it can be seen the transport and envelope fields are always the same, while the field named as “target” and “origin” can change depending on what agent started the request or what agent is able to serve the suitable response to the request (there are certain patterns, though. In a simple service request, the JSON request that arrives to the mote with the suitable sensor will have its MAC address and its name as the values of the target field, and the MAC address and the name of the Broker agent as the content in the origin field. In a composed service request, the Orchestrator agent appears as the content of target or origin fields). The operation and the parameters fields will vary depending on the service requested and the parameters used to have a more accurate response.

The JSON response message will have a very similar layout, being the only change that a new field named “result” will be placed under the parameters of the requested service:

```
{
      "transport": "j2me.radiogram",
      "envelope": "JSON-2.0",
      "target": "<IP or MAC address>/<name of the destination agent ",
      "origin": "<IP or MAC address>/<name of the source agent>",
      {
            "operation": "<operation name>",
            "parameters": [ < parameters or void, if there are none> ]
            "result": "<result of the operation>"
      }
}
```

Last but not least, when an alarm is requested, the answer brought to the final user via ESB will be a JSON message that looks like this:

```
{
      "transport": "j2me.radiogram",
      "envelope": "JSON-2.0",
      "target": "<IP or MAC address>/<name of the destination agent ",
      "origin": "<MAC address>/</nSOMOrchestratorAlarmsAgent",
      {
            "operation": "isAlarmed",
            "parameters": [ < the value that has triggered the alarm> ]
            "result": "<String representation of void object>"
      }
}
```

4.3 Analysis of the validation scenario results.

Once the scenario was mounted and the agents uploaded to the motes, several performance tests were carried out. The results, according to what services and features were tested, are offered here:

4.3.1 Setup of the Wireless Sensor Network and node reset.

What was measure here was how long it would take for the whole Wireless Sensor Network to be set up and have it fully functional. In order to perform the test, all the required nodes and the Zephyr belt were turned off to begin with, and they were progressively turned on. Banal as it may seem, it is a delicate procedure because the nodes cannot be reset or turned on at the same time, or without any order: obviously, if the Broker is the last node to be reset or turned on, the Wireless Sensor Network will not work at all because not a single service will be registered and all the requests will be systematically dropped. To have a successful setup, the mote with the Broker agent deployed must be the first one to be turned on or reset; after it, it is advisable to turn on the Zephyr belt first and the mote with the Zephyr agent uploaded right after that. Then the mote with the Orchestrator and the Orchestrator Alarms agents must be reset, and finally the motes with the temperature agents.

There are some other considerations that have to be made: the mote with the Orchestrator Alarms agent deployed (which will usually have the Orchestrator agent deployed as well) is the noisiest of the network by far. This is so because Orchestrator Alarms agent does not wait to be invoked by a user, but every five seconds (or in case of heart rate, one second) is asking the WSN for the data that it requires in order to make sure of the existence –or inexistence- of a value out of the bounds marked by the thresholds. Another troublesome node is the one with the Zephyr agent deployed; while this node does not add much data to the WSN, if the agent fails to be registered a lengthy series of actions have to be done to give it a try again (turning off the Zephyr belt, waiting around 10 seconds, turning on the belt again and finally resetting the mote) due to the behaviour of Bluetooth connections, thus adding a considerable amount of time in the network setup.

First, a chart containing how many milliseconds it took to have the WSN fully deployed is offered. In order to have reliable results each test was made by making twenty five measures. The results are as follows:

Requests	Setup
Request No. 1	89313
Request No. 2	78726
Request No. 3	101845
Request No. 4	52984
Request No. 5	212389
Request No. 6	140989
Request No. 7	59263
Request No. 8	80776
Request No. 9	54229
Request No. 10	59075
Request No. 11	86831
Request No. 12	143267
Request No. 13	56395
Request No. 14	267687
Request No. 15	125230
Request No. 16	120996
Request No. 17	126791
Request No. 18	90227

Request No. 19	109396
Request No. 20	153737
Request No. 21	252957
Request No. 22	138138
Request No. 23	91257
Request No. 24	86813
Request No. 25	58034

Chart 35: measures obtained for WSN setup.

Other interesting values obtained were the average (92337.726 milliseconds) and the median periods of time (91257 milliseconds). The small but significant difference between the average and the median values is indicating that there is certain heterogeneity with the obtained results, and actually, there are only two readings that are around 90 seconds; the disparity among them is widespread. This is due to the fact that several attempts were conditioned by having had an agent failing to register their first time, and therefore the required time at that measure soared, especially if that issue had happened with the Zephyr agent, or if the Orchestrator Alarms agent was overwhelming the Wireless Sensor Network.

The heterogeneity of the obtained values can be observed in the next graph:

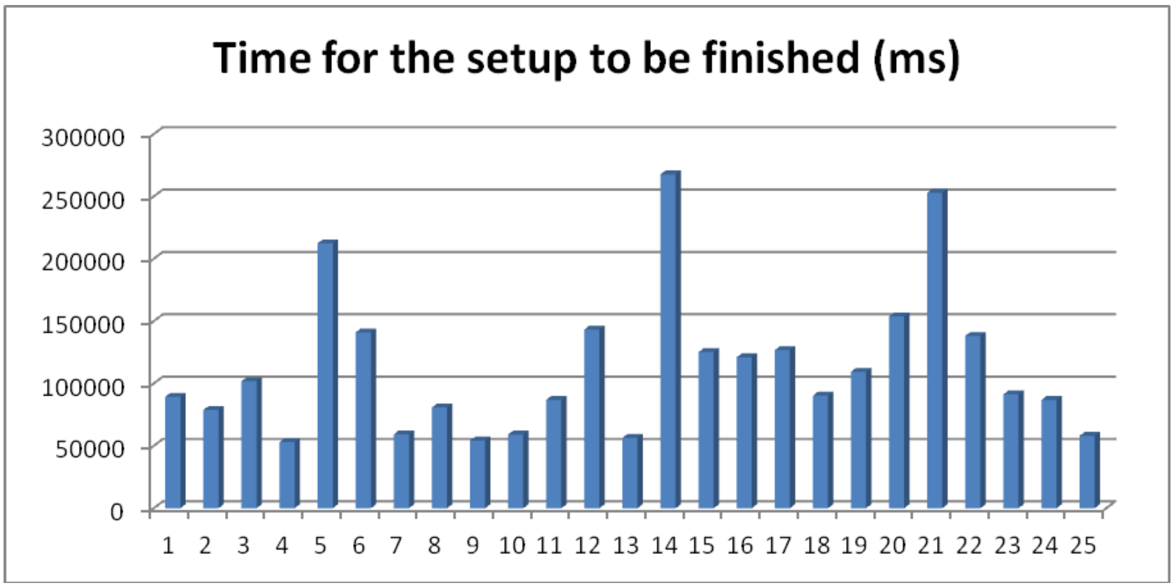


Figure 67: graph for the time used in setting up the mote (25 attempts).

4.3.2 Simple services analysis.

Once the Wireless Sensor Network has been setup, it is time to check how the provided services perform.

4.3.2.1 Analysis of Temperature1 service.

This is the first of three temperature readings that are going to be shown. Incidentally, the obtained results will not be the same for the three of them, because they have been placed under different conditions. The mote with the temperature agent named Temperature1 was the furthest away from the Broker; it was wanted to know how this would affect the communications.

For starters, in order to have 25 successfully answered requests, a total of 27 had to be made. This failure rate, as low as it may be (2 out of 27 attempts), is the highest of the three temperature agents tested. The other 25 requests were attended under this time values:

Request	Temp 1: time for the request to be responded (ms)
Request No. 1	813
Request No. 2	702
Request No. 3	821
Request No. 4	640
Request No. 5	752
Request No. 6	976
Request No. 7	620
Request No. 8	658
Request No. 9	788
Request No. 10	1103
Request No. 11	742
Request No. 12	838
Request No. 13	758
Request No. 14	682
Request No. 15	702
Request No. 16	1050
Request No. 17	638
Request No. 18	874
Request No. 19	731
Request No. 20	594
Request No. 21	708
Request No. 22	936
Request No. 23	720
Request No. 24	726
Request No. 25	681

Chart 36: measures obtained for Temperature1 service.

Other interesting values obtained were the average (751.207 milliseconds) and the median times (731 milliseconds). The disparity between the average value and the median, albeit small, is still significant. This is due to the fact that some of the requests took a longer than usual time to be attended as it can be seen on the next graph, made with the text from the previous chart:

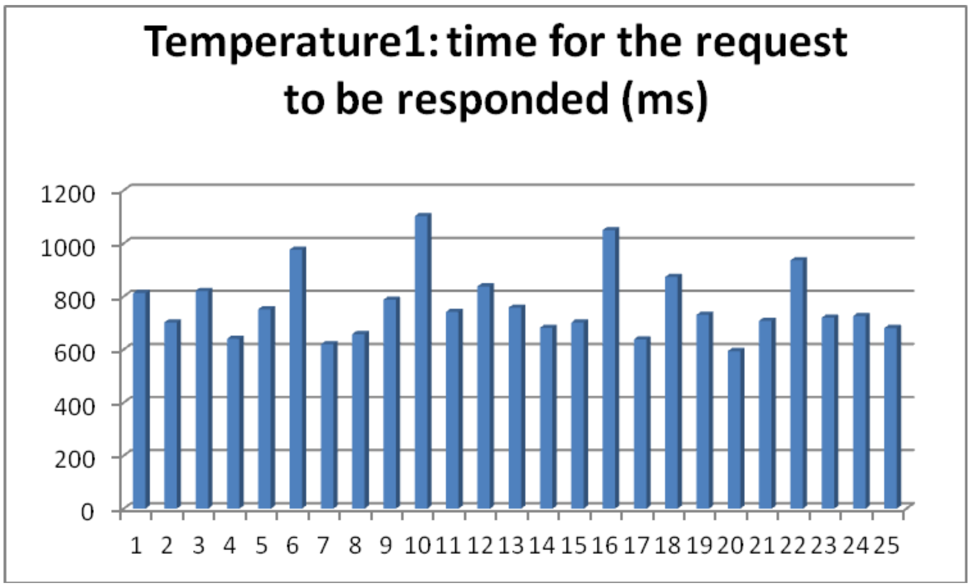


Figure 68: graph for the time used in completing Temperature1 request (25 attempts).

4.3.2.2 Analysis of Temperature2 service.

The mote with the temperature agent named Temperature2 deployed was under different conditions than the former: it was relatively near the mote with the Broker agent, but its service was requested by the Orchestrator Alarms agent for the environmental temperature every little time, so it was under an acceptable but constant stress. Apparently, though, it did not affect the performance of the agent because only one request failed to be attended (1 out of 26 attempts). Surprisingly, the obtained values were slightly better than with Temperature1, further away but with no other requests that the ones made by the user:

Request	Temp 2: time for the request to be responded (ms)
Request No. 1	801
Request No. 2	532
Request No. 3	619
Request No. 4	685
Request No. 5	724
Request No. 6	746
Request No. 7	1133
Request No. 8	1048
Request No. 9	550
Request No. 10	728
Request No. 11	841
Request No. 12	670
Request No. 13	692
Request No. 14	669
Request No. 15	850
Request No. 16	806
Request No. 17	824

Request No. 18	667
Request No. 19	914
Request No. 20	664
Request No. 21	713
Request No. 22	656
Request No. 23	702
Request No. 24	692
Request No. 25	591

Chart 37: measures obtained for Temperature2 service.

Other important data obtained were the average (718.515 milliseconds) and the median values (702 milliseconds). It is interesting to note that the difference between the average and the median data is lower than it was with Temperature1 service. This is pointing out at the fact that the data are more regular, and also they are confined in a narrower range of values. That is the impression that is offered at the sight of the next graph:

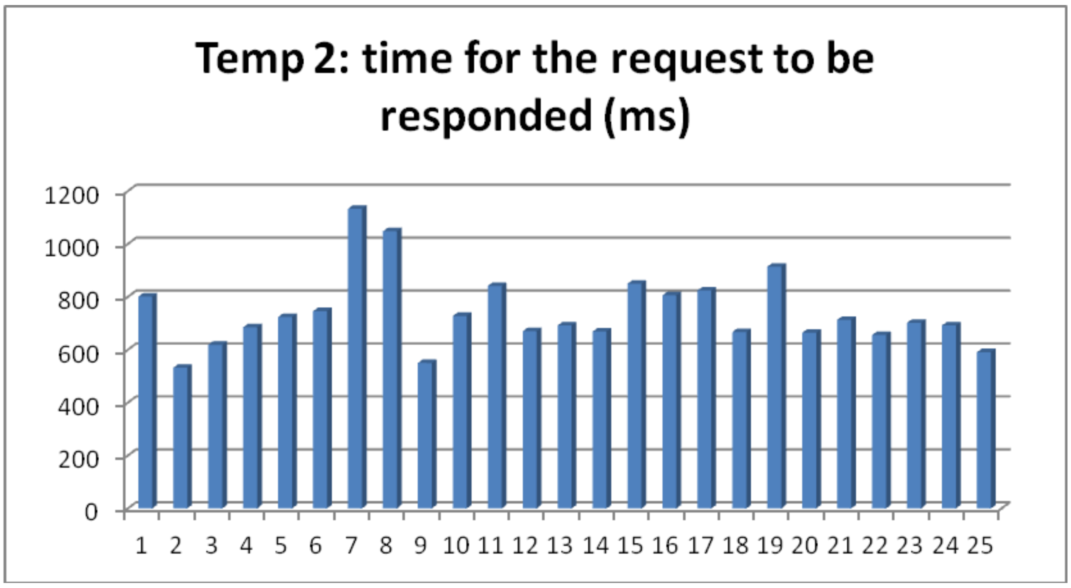


Figure 69: graph for the time used in completing Temperature2 request (25 attempts).

4.3.2.3 Analysis of Temperature3 service.

The mote with the Temperature3 agent deployed on its hardware was given a third different environment: it was at a closer distance than the mote with Temperature1 agent deployed, but further away from the Broker agent than Temperature2 mote was. However, it did not attend any other request than the ones that were made during this testing session. Given that this was an agent with quite favourable conditions, it should come not as a surprise that the 25 requests were attended without any failure (failure rate: 0/25 attempts). Along with it, the results obtained from the tests, as long as required time is concerned, were like this:

Request	Temp 3: time for the request to be responded (ms)
Request No. 1	603
Request No. 2	741
Request No. 3	681
Request No. 4	671
Request No. 5	702
Request No. 6	669
Request No. 7	728
Request No. 8	607
Request No. 9	753
Request No. 10	783
Request No. 11	710
Request No. 12	750
Request No. 13	857
Request No. 14	665
Request No. 15	702
Request No. 16	758
Request No. 17	702
Request No. 18	760
Request No. 19	641
Request No. 20	656
Request No. 21	742
Request No. 22	765
Request No. 23	678
Request No. 24	761
Request No. 25	651

Chart 38: measures obtained for Temperature3 service.

Other noteworthy values obtained were the average (704.760 milliseconds) and the median values (702 milliseconds). These values are very interesting when compared to the ones obtained from Temperature1 and Temperature2 agents: not only they are slightly lower, but also the average and the median values have reduced their difference extraordinarily; clearly, having a mote without any issue related with the power of the radio signal, and without the disturbance of having to attend requests from two sides (a human user and an inner agent) pays off when its performance is considered.

A graph with the results shown before is on next page:

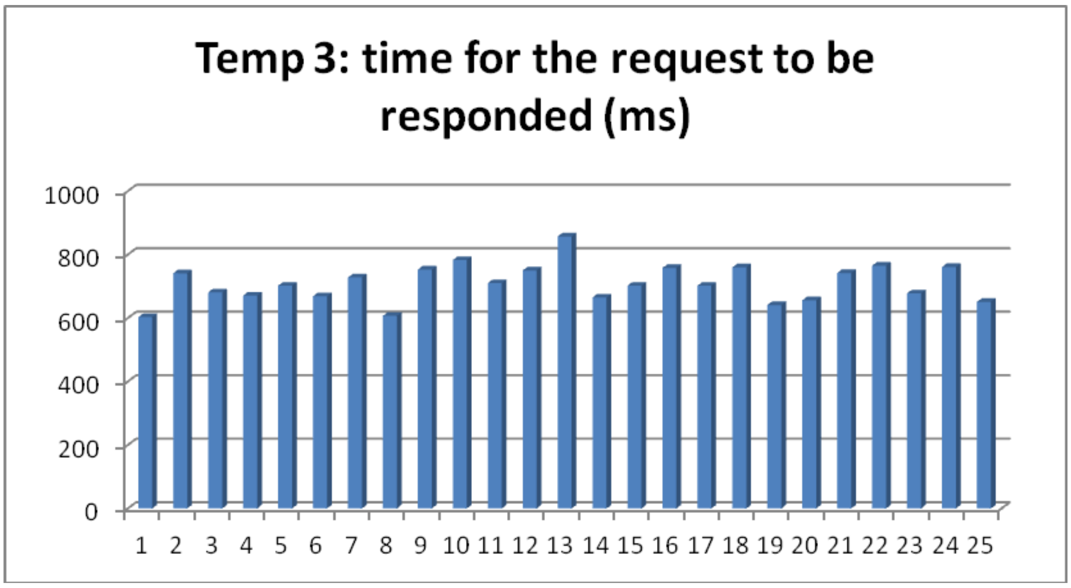


Figure 70: graph for the time used in completing Temperature3 request (25 attempts).

4.3.2.4 Analysis of body temperature service.

This service has a strong difference with the others, despite being a simple service like them: it is not depending on a “local” sensor to retrieve the data, but it has them sent from the Zephyr belt via Bluetooth transmission. Since the Zephyr device will transmit data every second, a performance less appealing than the ones of the temperature agents should be expected. But actually, although its performance is lower when compared to the other motes, it is not too much lower.

In order to have 25 requests successfully answered, 27 had to be taken (error rate: 2/27 attempts), just like it happened with the furthest away mote, the one with the Temperature1 agent deployed. The obtained results are as follows:

Request	body temp.: time for the request to be responded (ms)
Request No. 1	603
Request No. 2	741
Request No. 3	681
Request No. 4	671
Request No. 5	702
Request No. 6	669
Request No. 7	728
Request No. 8	607
Request No. 9	753
Request No. 10	783
Request No. 11	710
Request No. 12	750
Request No. 13	857
Request No. 14	665
Request No. 15	702

Request No. 16	758
Request No. 17	702
Request No. 18	760
Request No. 19	641
Request No. 20	656
Request No. 21	742
Request No. 22	765
Request No. 23	678
Request No. 24	761
Request No. 25	651

Chart 39: measures obtained for Temperature3 service.

Considering that the average (775,296 milliseconds) and the median times (755 milliseconds) are higher than in the other cases, and the difference between the average and the median value increases –thus, the results are less homogeneous than before- the expectations about this agent are confirmed. However, this agent has to be given credit for not lowering the performance that much, especially given that is dependent on an external device to harvest the data from the environment.

A graph with the already represented data is below:

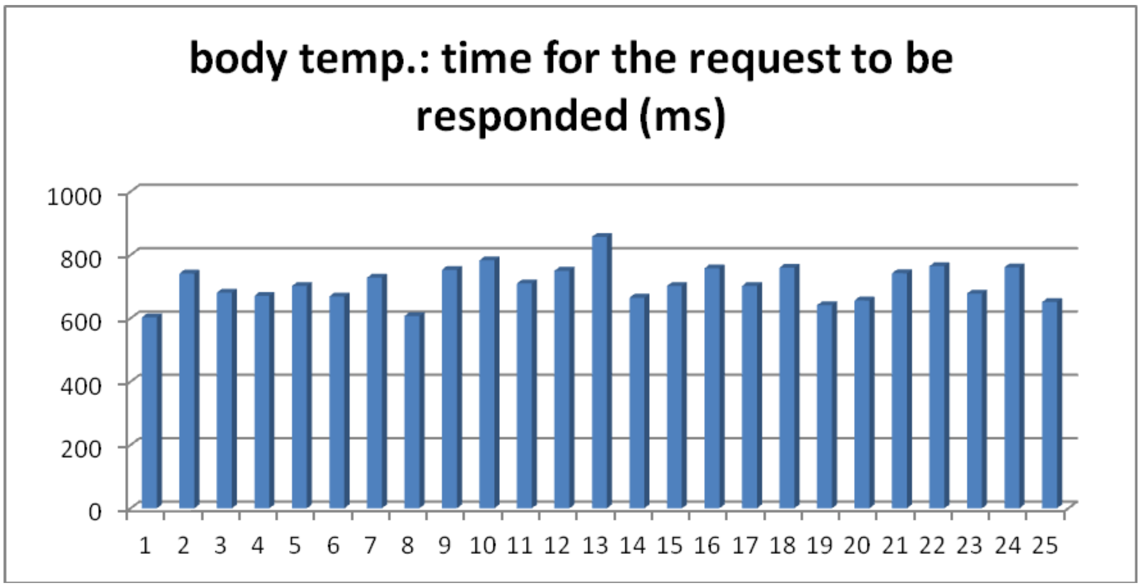


Figure 71: graph for the time used in completing Body Temperature request (25 attempts).

4.3.2.5 Conclusions about performance in simple services.

From the data obtained before, there are some conclusions that can be inferred.

- Apparently, the distance (and therefore, the strength of the signal that is communicating with the motes) is more of a issue than a moderate load of traffic when requests coming from a user have to be responded. The requests will fail a little oftener and the ones that are delivered will be slightly slower and less reliable if one mote is too far away. Obviously, the lesser traffic a mote has to deal with and the nearer it is (but not so near to have interference phenomena), the better its performance will be.

- When an external factor is put into use in a Wireless Sensor Network (in our case, a Bluetooth-enabled belt), it is quite probable that it will lower the performance of the WSN element it attaches to, because it will make it dependent on the external device in its data deliveries. Nevertheless, if it has to be done, Sun SPOT motes have proven not to crash easily.

It has to be born in mind that all the tests that are shown here were done with the motes fully charged, so battery levels should not be an issue when comparing performances among services.

4.3.3 Composed services analysis.

To offer a wider view of the general performance of the system, and in order to have information about how the network works with different services, the same analysis procedure has been applied on two of all the composed services of the system.

4.3.3.1 Analysis of injury prevention service.

This service fares the worst in terms of performance due to two reasons: firstly, it requires the retrieval of three pieces of information to be fully and successfully delivered; secondly, the service is requested under not so favourable conditions, with an element of the Wireless Sensor Network (the Orchestrator Alarms agent deployed altogether with the Orchestrator node) sending requests for data in a fast pace. Its reliability is affected as well: out of 28 requests done, three failed (error rate: 3/28 attempts); it is the worst result of all the services tested here.

Consequently, the time required to serve this service is way longer than it was with the simple services (even when all the required times of the simple services are summed), as it can be learnt from the next chart:

Request	Injury prev.: time for the request to be responded (ms)
Request No. 1	11408
Request No. 2	4637
Request No. 3	18009
Request No. 4	11214
Request No. 5	4466
Request No. 6	11267
Request No. 7	18121
Request No. 8	4691
Request No. 9	4596
Request No. 10	4381
Request No. 11	11184
Request No. 12	11269

Request No. 13	17977
Request No. 14	17821
Request No. 15	18159
Request No. 16	11123
Request No. 17	4635
Request No. 18	18185
Request No. 19	11654
Request No. 20	11182
Request No. 21	16840
Request No. 22	11213
Request No. 23	16894
Request No. 24	4538
Request No. 25	11257

Chart 40: measures obtained for injury prevention service.

As for the average and the median times, they are 8702.681 milliseconds and 11257 milliseconds (this is the only tested service where the median value was bigger than the average one).There are two important facts that must be taken into account about this measures: firstly, the results obtained are, as it was mentioned before, much worse than in simple services. Apart from that, a pattern can be learnt from here: all the data collected are showing periods of time around 4.5, 11 and 18 seconds. It is probably due to the fact that the Broker and the Orchestrator agent are competing against the Orchestrator Alarms agent to get the data, and in this competence they can be very successful (the three pieces of information for the injury prevention service will be obtained before the data for the alarm checking procedure, thus taking for the service around 4.5 seconds to be served), mildly successful (the Broker and the Orchestrator agent must wait for the Orchestrator agent to fish one data request, lasting a time long enough to serve the injury prevention service in around 11 seconds), or unsuccessful (the Broker and the Orchestrator agent have to wait even more Orchestrator Alarm agent requests, thus increasing the required time to around 18 seconds).

This fact can be seen on the graph below:

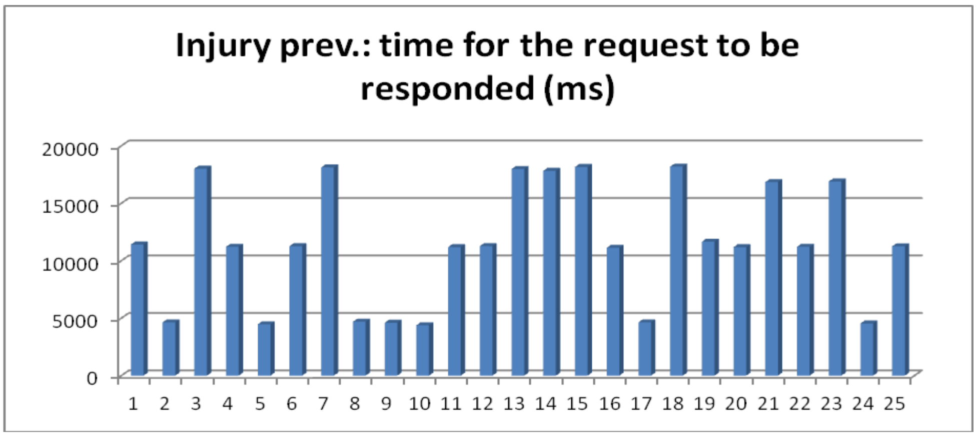


Figure 72: graph for the time used in completing Injury prevention request (25 attempts).

4.3.3.2 Analysis of temperature control service.

This composed Service requires less information pieces to be composed, and therefore its performance is better than in the case before. Its reliability is somewhat better too: out of 27 requests, 2 failed to provide the service (error rate: 2/27 attempts).

As it can be seen in the next chart, the phenomenon that was happening before about readings collected around very specific values is reproduced here, albeit at a different, lower scale:

Request	Temp control: time for the request to be responded (ms)
Request No. 1	3309
Request No. 2	10105
Request No. 3	3122
Request No. 4	9916
Request No. 5	3303
Request No. 6	3390
Request No. 7	15540
Request No. 8	3236
Request No. 9	3205
Request No. 10	3239
Request No. 11	3200
Request No. 12	9932
Request No. 13	3195
Request No. 14	3177
Request No. 15	3414
Request No. 16	9965
Request No. 17	15342
Request No. 18	9974
Request No. 19	15626
Request No. 20	3775
Request No. 21	9922
Request No. 22	15870
Request No. 23	3672
Request No. 24	3496
Request No. 25	9924

Chart 41: measures obtained for temperature control service.

As for the average and the median times, they are 4839.654 milliseconds and 3672 milliseconds respectively. Since now only two simple services are required to compose the third one, the times around the services are delivered are 3, 10 and 15 seconds. All these facts can be seen in the next graph as well:

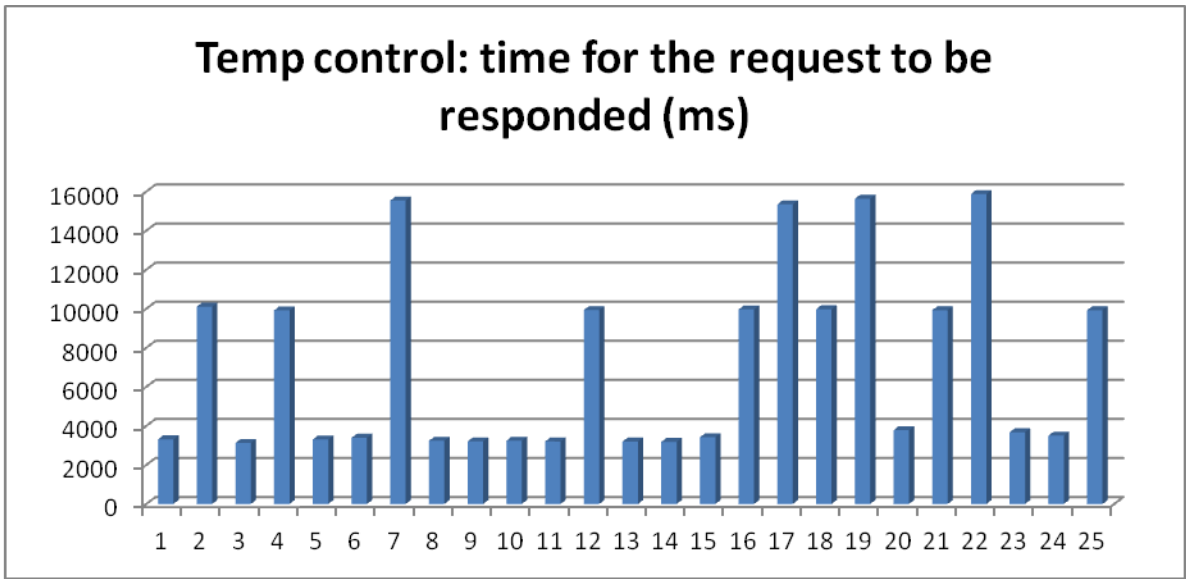


Figure 73: graph for the time used in completing temperature control request (25 attempts).

4.3.3.3 Conclusions about performance in composed services.

As it was done before, conclusions regarding how composed services are offered in our system using nSOM semantic middleware architecture are presented:

- Composed services take heavy punishment from the almost constant activity of the Orchestrator Alarms agent: since they require a lot of messages until finally completing their tasks (especially if many pieces of simple information are required), they are prone to suffer from delays at any of the links needed to have a fully functional chain of requests and responses, both inside and outside the domain of the Wireless Sensor Network.

- The performance of a composed service, albeit much lower than that of a simple service, is at least fairly predictable: depending on how fast the requests for simple data were attended the result will be obtained around very specific values

4.3.4 Reset time analysis.

The last test that is going to be offered is regarding the time required for an agent to get successfully registered at the Broker agent. As the reliability of Wireless Sensor Networks is one of their most important features to be considered, it seems interesting to know how long would take an agent to register itself again if the mote it is deployed on has gone down for any reason (data floods, energy depletion, etc.). Another 25 attempts were done at this test, and Temperature2 agent was used for this process. The results obtained were as follows:

Request	Time for an agent to get re-registered (ms)
Attempt No. 1	7309
Attempt No. 2	7375
Attempt No. 3	7444

Attempt No. 4	7307
Attempt No. 5	7205
Attempt No. 6	7184
Attempt No. 7	7246
Attempt No. 8	7298
Attempt No. 9	7178
Attempt No. 10	7367
Attempt No. 11	7457
Attempt No. 12	7448
Attempt No. 13	7376
Attempt No. 14	7371
Attempt No. 15	7364
Attempt No. 16	7275
Attempt No. 17	7305
Attempt No. 18	7353
Attempt No. 19	7410
Attempt No. 20	7352
Attempt No. 21	7252
Attempt No. 22	7288
Attempt No. 23	7315
Attempt No. 24	7224
Attempt No. 25	7226

Chart 42: measures obtained for re-registering an agent.

The data obtained are very homogeneous (in fact, the average and the median values are 7316.298 milliseconds and 7309 milliseconds, proportionally, the least differentiated of all the tests done), although somehow disappointing because the required time seems to be longer that what would be considered as desirable (results around 3 or 4 seconds were expected).

The data represented in the chart above these Lines has been poured into a graph below:

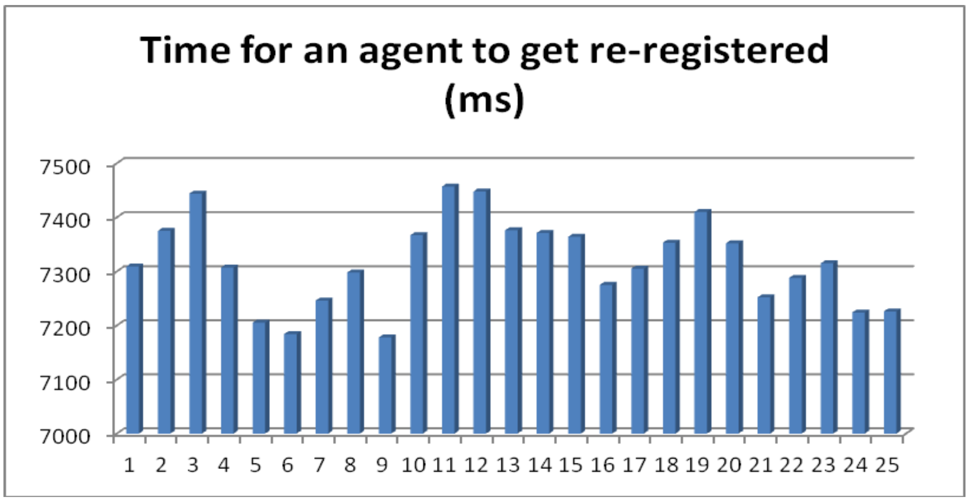


Figure 74: graph for the time used in completing re-registering (25 attempts).

Chapter 5

Conclusions and future works

5.1 Conclusions.

The content of this report has been primarily focused on three major topics: a) A study on the state of the art of the current situation of ubiquitous middleware architectures, as a starting point to get an idea of what problems can be found when developing middleware under this conditions, b) A formal specification of a ubiquitous, pervasive middleware architecture with a description of all its elements, along with what functionalities can be expected and how they operate, and c) A third stage where this architecture is put to an actual use under a real system, obtaining a collection of data that is studied so as to extract information from it.

In order to conclude this report in a more compelling way, a brief analysis of each of the already presented chapters is going to be made here to infer several conclusions at the sight of it from a holistic point of view, that will make possible to better evaluate the work done in this Final Degree Dissertation, along with several proposals that would be interesting to be taken into consideration for future developments on this technology.

To begin with, the topics that have played a key role in this Final Degree Dissertation were first introduced in Chapter 1: the foundations of ubiquitous computing, along with all the other names that it has been given, the concepts that were linked to it, and the challenges inherently associated with developing software for this environment. Besides, the motivations, objectives and contributions quoted in this Final Degree Dissertation, the structure of the whole report and the technological framework it is encased in were mentioned as well.

Being the main topic this report is about semantic, ubiquitous and pervasive middleware, a compelling description of the state of the art in ubiquitous middleware was offered in Chapter 2. All the troublesome features that had been hinted before were fully displayed here. First of all, a reasoning of why is a middleware architecture layer required was made, stating that middleware architectures dated decades before this Final Degree Dissertation, and given that one of them would be needed in a development of ubiquitous nature, why this middleware architecture could not be dealt with in a similar way as if it was a more conventional system. Although the objectives and the solutions seemed to be the same that were used on regular environments, realistically, middleware developments must be adapted to the special requirements of wearable and pervasive computing.

Once this was learnt, a collection of legacy solutions in ubiquitous middleware was offered. Although many of the developments have important and somewhat common flaws (not ubiquitous enough, limited to a small scenario, not complete hardware insulators) they have to be given credit for being serious attempts at giving a solution to the challenges faced under the conditions that interest this Final Degree Dissertation. Plus, generally faulty as they may be, their failures have been a source for feedback and improvement of later developments, and it does not look as a coincidence that these new developments started appearing when new approaches were tried towards ubiquitous computing. These new approaches offer general purposes, objectives and new paradigms that ought to be taken into account when developing a middleware architecture; they provide new points of view and solve some of the issues presented before (software-centric architectures do not consider hardware that much anymore, the Internet of Things -or IoT- gives a final goal for applications of where to point, etc.), although as it has been seen, they are not perfect and have their own issues, as the former middleware architectures based on earlier paradigms had.

It also has become clear in this chapter that there is a wide variety of motes to choose from when thinking of a ubiquitous system –for a wider knowledge, one of the annexes shows a complete list of all the motes and gateway motes that are being marketed-; while not mandatory by any means, Wireless Sensor Networks are usually a key component for middleware architectures, and because WSNs are usually composed by intercommunication among a network of motes (regardless of what may be at the end of the system, like an user-carried computer or a BAN/PAN), choosing a particular model of a mote is a critical task for the whole ubiquitous system that is going to be deployed. Among all of them, Sun SPOT motes stood out for several reasons: their RAM and ROM capabilities were impressive and the usage of the Java Virtual Machine as the operating system guarantees a low footprint of the code in use.

Finally, some of the open issues pending on hardware platforms were mentioned too. Basically, the development of more efficient and capable Wireless Sensor Networks is strongly dependent on the development of motes and their technologies, so any breakthrough in these fields will have a considerable impact on future implementations. Furthermore, it is pointed out that several services that are almost a given in conventional environment (Quality of Service, security) are underdeveloped or simply nonexistent in Wireless Sensor Networks and middleware architectures of this kind; an effort must be done to solve this issue.

Next chapter was crucial to the introduction to an example of a service and ubiquitous oriented middleware. A general overview of the nSOM semantic middleware architecture was first offered, with all its different components and all the messages that are used whenever a communication is established (or finished). Before talking about its functionalities, the components or this architecture had to be introduced. nSOM is an agent-oriented semantic architecture, so its main components from a software-centric point of view (from the hardware perspective, the usual Wireless Sensor Network can be expected, along with some other parts that are going to be specifically mounted for the system validation) are agents; these agents can be uploaded to the motes at will, and it is not unusual (in some cases it is even mandatory) to have more than one agent loaded into each of the motes. Only the base station has not agents, although it is playing another role that makes it almost as if there was one of them (performing bridging functionalities between the ubiquitous environment and the ESB present in a Personal Computer).

After all its components were described, it was time to show what they could offer altogether. Apart from the readings that could be obtained from the data harvested by the sensors (environmental temperature) or converted from the Zephyr belt (heart rate, breathing rate, body temperature), those readings could be combined into composed services that would result in a third service whose value can be inferred from the combined values of the previous readings. This value obtained from a composed service would be the response of a request undertaken by the final user, who must remain oblivious to the fact that there are no physical sensors measuring any data (concept of sensor virtualization) despite requesting an actual value.

Lastly, once all this aspects were clarified, a system was built within the technological framework that the Lifewear project could offer. After reasoning what kind of scenario the system would be built on, the user cases found were analyzed, along with the actors playing an active role in each of them. As it has been read, they were all based on the capabilities that were described before in the previous chapter. A description of the validation scenario is the next point of the chapter; the elements that are taking part of it, and the places they are going to be located, are told in an accurate way. From this deployed architecture a certain amount of data was measured, and from the results that were obtained, once they were processed into charts and graphics, several tendencies and conclusions were given.

5.2 Future works.

Although nSOM middleware architecture has proved to have a high level of maturity through the Lifewear project, and has become noteworthy in the fields of ubiquitous computing and semantic, pervasive middleware by its own right, there are still several improvements that could be interesting to tackle in future versions, judging from the results that have been obtained in the real scenario:

- A stronger focus on **node scalability** would be desirable. If all the motes that are used in this project are counted (plus the two Sun SPOT motes and the base station from Tecnalia) we have nine motes (three indoor temperature nodes, two Bluetooth-related motes, one Broker, one Orchestrator –belonging to the deployment to the EUITT-, an outdoor temperature mote, an indoor luminosity mote –belonging to the deployment of Tecnalia-) and two base stations (one for the deployment done with the infrastructure of EUITT and another for Tecnalia´s). For a relatively small indoor scenario is acceptable, but if a more ambitious approach was tried for another purpose different from a gymnasium (for example, a whole building instead of a room, a wide outdoor area, etc.), an extra effort must be placed in having a perfectly functional system with, at least, dozens of motes. Nevertheless, the variety of challenges that have been overcome under the described deployment (fast-paced data requests, simultaneous service requests, coexistence with other architecture with the same kind of motes, Bluetooth data conversion) is remarkable and proves that the nSOM architecture has a lot of potential for more complex developments.

- Sophisticated mechanisms to improve the **reliability of the system**. Self-healing and self-recovery systems would come in handy for future developments, especially if a higher number of motes (and therefore, a higher number of chances for the failure of one of them) is going to be part of the Wireless Sensor Network. There are several ways to get through this task: nodes can check among them if the expected functionalities of the others are being made; should there be any trouble, another mote could assume the duties to be undertaken of the mode that has failed to do so. Also, a model where Broker or Orchestrator functionalities are decentralized from one to several motes could make the system more reliable (if one of the motes that is part of a “Broker federation” runs out of battery, the other motes can still perform much or the majority of their work), at the expense of having several motes to fulfil one particular task instead of just one.

- If this architecture is going to be used in a wider area, where there are many rooms and many places, a **location system** could be useful. Although right now it is possible to know where every mote is by using very basic procedures (if someone has attached a temperature mote to the wall of a particular room, then this person knows which room the mote is), if the environment that is going to receive the Wireless Sensor network, or the WSN itself, are very wide, taking notes about where the dozens or even hundreds of nodes are is clearly not an advisable idea. It would be more sensible to obtain the information whenever a request is done as part of the JSON answer, to have this data in a more precise way (asking for the temperature of one spot of one room, for example) or even requesting the accurate location of a mote for future data retrieval.

- Finally, a **software management application** to better handle code from/to the motes would be another improvement to think of. Currently, the portability of the Java classes that are uploaded to a mote is very limited; an IDE is required to modify them, and in order to upload a different version of that code, or software with different functionalities, an uncomfortable procedure of looking through the projects of the IDE has to be done. It seems like a better solution if a portable application with a user-friendly interface could do that procedure and the file uploading without having to worry about having an IDE. It could be done by offering several options, like the available agents to be uploaded (Temperature, Orchestrator) or different software versions. Even minor changes (for example, lowering the body temperature maximum acceptable value) could be managed from the interface of the application, making it more attractive for a final user or a developer.

Annex I

APIs of the nSOM components

To this day, nSOM middleware architecture makes use of 29 java classes; since providing the Javadoc of all of them would be too impractical and could make the reader lose the perspective of the agents involved in the communications only the most important classes are shown here:

Classes related to the Broker agent:

runSun SPOT.java

```
public class runSun SPOT
extends MIDlet
```

The startApp method of this class is called by the VM to start the application. The manifest specifies this class as MIDlet-1, which means it will be selected for execution.

Constructor Summary
runSun SPOT()

Method Summary	
protected void	destroyApp(boolean unconditional) Method used to finish the application.
protected void	pauseApp() Method used to pause the applications, it is not currently called by the Squawk Virtual Machine.
protected void	startApp() Method used to start the application and execute all the uploaded code.
void	switchPressed(ISwitch sw) Method that defines the actions to be taken when a switch is pressed.
void	switchReleased(ISwitch sw) Method that defines the actions to be taken when a switch is released.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

runSun SPOT

```
public runSun SPOT()
```

Method Detail

startApp

```
protected void startApp()
                 throws MIDletStateChangeException
```

Method used to start the application and execute all the uploaded code.

Throws:

`MIDletStateChangeException`

---

pauseApp

```
protected void pauseApp()
```

Method used to pause the applications, it is not currently called by the Squawk Virtual Machine.

---

destroyApp

```
protected void destroyApp(boolean unconditional)
                   throws MIDletStateChangeException
```

Method used to finish the application.

Parameters:

`unconditional` - not currently used by the application.

Throws:

`MIDletStateChangeException` - exception thrown if there is an illegal change of state.

---

switchPressed

```
public void switchPressed(ISwitch sw)
```

Method that defines the actions to be taken when a switch is pressed.

Parameters:

`sw` - object from the ISwitch interface

---

switchReleased

```
public void switchReleased(ISwitch sw)
```

Method that defines the actions to be taken when a switch is released.

Parameters:

`sw` - object from the ISwitch interface

nSOMBrokerAgent.java

```
public class nSOMBrokerAgent
extends java.lang.Object
implements nSOMAgent
```

Project: Lifewear

Constructor Summary
`nSOMBrokerAgent()`

Method Summary	
`void`	`connectionReset(java.lang.String dottedAddress)` Performs the load process of the nSOMBrokerAgent in the Service Execution Platform.
`java.lang.String`	`getnSOMAgentDescription()` Obtain the nSOMBrokerAgentDescription.
`nSOMServiceAgentObject`	`getServiceAgentObject()` Obtain the service description object for this nSOMLightAgent.
`void`	`load(nSOMContext nSOMContext)` Performs the load process of the nSOMBrokerAgent in the Service Execution Platform.
`void`	`receptacle(ServiceContext serviceResponse)` Implements the receptacle for service responses of the nSOMBrokerAgent.
`void`	`run()`

	Performs the run process of the nSOMBrokerAgent.
void	`serviceInvocation(ServiceContext serviceInvocation)` Implements the receptacle for service invocation of the nSOMBrokerAgent.
void	`startSenderThreadPort52()` Method that will control the radio connections and communications through port number 52 (composed services requests).
void	`startSenderThreadPort54()` Method that will control the radio connections and communications through port number 54 (alarm requests).
void	`startSenderThreadPort67()` Method that will control the radio connections and communications through port number 67 (simple services requests).
void	`stop()` Performs the stop process of the nSOMBrokerAgent.
void	`unload()` Performs the unload process of the nSOMBrokerAgent in the Service Execution Platform.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

nSOMBrokerAgent

`public nSOMBrokerAgent()`

Method Detail

load

`public void load(nSOMContext nSOMContext)`

Performs the load process of the nSOMBrokerAgent in the Service Execution Platform.

Specified by:

`load` in interface `nSOMAgent`

Parameters:

`Context` - provided by the nSOMContainer

---

connectionReset

`public void connectionReset(java.lang.String dottedAddress)`

Performs the load process of the nSOMBrokerAgent in the Service Execution Platform.

Parameters:

`Context` - provided by the nSOMContainer

---

run

`public void run()`

Performs the run process of the nSOMBrokerAgent. It is listening for data from the DiYSE cloud

Specified by:

`run` in interface `nSOMAgent`

---

stop

`public void stop()`

Performs the stop process of the nSOMBrokerAgent.

Specified by:

`stop` in interface `nSOMAgent`

---

unload

`public void unload()`

Performs the unload process of the nSOMBrokerAgent in the Service Execution Platform.

Specified by:

`unload` in interface `nSOMAgent`

---

serviceInvocation

`public void serviceInvocation(ServiceContext serviceInvocation)`

Implements the receptacle for service invocation of the nSOMBrokerAgent. This method is invoked to pass the service invocations to the Broker agent.

Parameters:

`ServiceContext` - of the service invocation

---

receptacle

`public void receptacle(ServiceContext serviceResponse)`

Implements the receptacle for service responses of the nSOMBrokerAgent. This method is invoked to pass the service responses to the Broker agent.

Specified by:

`receptacle` in interface `nSOMAgent`

Parameters:

`ServiceContext` - of the service response

---

getServiceAgentObject

`public nSOMServiceAgentObject getServiceAgentObject()`

Obtain the service description object for this nSOMLightAgent.

Specified by:

`getServiceAgentObject` in interface `nSOMAgent`

Returns:

nSOMServiceAgentObject of the nSOMLightAgent

---

getnSOMAgentDescription

`public java.lang.String getnSOMAgentDescription()`

Obtain the nSOMBrokerAgentDescription. This method provides the name of the broker agent, which have been invoked.

Specified by:

`getnSOMAgentDescription` in interface `nSOMAgent`

Returns:

String with the Broker agent description

---

startSenderThreadPort67

`public void startSenderThreadPort67()`

Method that will control the radio connections and communications through port number 67 (simple services requests).

---

startSenderThreadPort52

`public void startSenderThreadPort52()`

Method that will control the radio connections and communications through port number 52 (composed services requests).

---

startSenderThreadPort54

`public void startSenderThreadPort54()`

Method that will control the radio connections and communications through port number 54 (alarm requests).

Classes related to the Orchestrator agent:

nSOMOrchestratorAgent.java

```
public class nSOMOrchestratorAgent
extends java.lang.Object
implements nSOMAgent
```

Project: Lifewear

Field Summary	
static double	**MAX_BODY_TEMPERATURE** Variable used to have a value for maximum body temperature.
static int	**MAX_HEART_RATE** Variable used to have a value for maximum heart rate.
static double	**MAX_ROOM_TEMPERATURE** Variable used to have a value for maximum room temperature.
static double	**MIN_BODY_TEMPERATURE** Variable used to have a value for minimum body temperature.
static int	**MIN_HEART_RATE** Variable used to have a value for minimum heart rate.
static double	**MIN_ROOM_TEMPERATURE** Variable used to have a value for minimum room temperature.

`static int`	**`WARNING_MARGIN`** Variable used to have a margin value; it will be useful to define levels on the results of the composed services.
`static double`	**`WARNING_MARGIN_TEMP`** Variable used to have a margin value; it will be useful to define levels on the results of the composed services related with body temperature.

Constructor Summary
`nSOMOrchestratorAgent``()`   Constructor of the nSOMOrchestratorAgent Class.

Method Summary	
`double`	**`bluetoothRequest`**`(java.lang.String serviceInvocationSMD)` Method used to extract the value obtained from the Bluetooth-associated mote from a JSON answer.
`java.lang.String`	**`getInjuryPrevention`**`()` Method used to get a lecture of the risk of having a muscular breakdown while performing a workout.
`java.lang.String`	**`getnSOMAgentDescription`**`()` Obtain the nSOMAgentDescription.
`nSOMServiceAgentObject`	**`getServiceAgentObject`**`()` Obtain the service description object for this nSOMTemperatureAgent.
`java.lang.String`	**`getTemperatureControl`**`()` Method used to get a lecture of the temperature levels while performing.
`java.lang.String`	**`getThermometer`**`()`
`void`	**`load`**`(nSOMContext nSOMContext)` Performs the load process of the nSOMTemperatureAgent in the Service Execution Platform.
`java.lang.String`	**`receiveDataRequestFromWSN`**`()` Process an incoming service request from the DiYSE cloud to the WSN,

	waiting the service response from the sensor network via Broker node
`void`	`receptacle(ServiceContext operationInvocation)` Implements the receptacle of the nSOMTemperatureAgent.
`void`	`run()` Performs the run process of the nSOMTemperatureAgent and the services registering in the Brokers.
`void`	`scanAlarms()`
`void`	`sendDataRequestToWSN(java.lang.String serviceInvocationSMD)` Process an incoming service request from the DiYSE cloud to the WSN, sending it to the sensor network via Broker node
`void`	`stop()` Performs the stop process of the nSOMTemperatureAgent and the services unregistering in the Brokers.
`double`	`tomarTemperatura(java.lang.String serviceInvocationSMD)` Method used to extract the temperature value from a JSON answer.
`void`	`unload()` Performs the unload process of the nSOMTemperatureAgent in the Service Execution Platform.
`java.lang.String`	`wirelessNetworkRequest(java.lang.String operationName)` Method used to perform requests on the network in order to obtain the required data for the composed services

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

MAX_ROOM_TEMPERATURE

`public static final double` **`MAX_ROOM_TEMPERATURE`**

Variable used to have a value for maximum room temperature.

See Also:

Constant Field Values

MIN_ROOM_TEMPERATURE

`public static final double` **`MIN_ROOM_TEMPERATURE`**

Variable used to have a value for minimum room temperature.

See Also:

Constant Field Values

MAX_BODY_TEMPERATURE

`public static final double` **`MAX_BODY_TEMPERATURE`**

Variable used to have a value for maximum body temperature.

See Also:

Constant Field Values

MIN_BODY_TEMPERATURE

`public static final double` **`MIN_BODY_TEMPERATURE`**

Variable used to have a value for minimum body temperature.

See Also:

Constant Field Values

WARNING_MARGIN

`public static final int` **`WARNING_MARGIN`**

Variable used to have a margin value; it will be useful to define levels on the results of the composed services.

See Also:

Constant Field Values

WARNING_MARGIN_TEMP

`public static final double` **`WARNING_MARGIN_TEMP`**

Variable used to have a margin value; it will be useful to define levels on the results of the composed services related with body temperature.

See Also:

Constant Field Values

MAX_HEART_RATE

`public static final int` **`MAX_HEART_RATE`**

Variable used to have a value for maximum heart rate.

See Also:

Constant Field Values

MIN_HEART_RATE

`public static final int` **`MIN_HEART_RATE`**

Variable used to have a value for minimum heart rate.

See Also:

Constant Field Values

Constructor Detail

nSOMOrchestratorAgent

`public` **`nSOMOrchestratorAgent`**`()`

Constructor of the nSOMOrchestratorAgent Class.

Method Detail

load

`public void` **`load`**`(nSOMContext nSOMContext)`

Performs the load process of the nSOMTemperatureAgent in the Service Execution Platform.

Specified by:

`load` in interface `nSOMAgent`

Parameters:

`Context` - provided by the nSOMContainer

---

run

`public void` **`run`**`()`

Performs the run process of the nSOMTemperatureAgent and the services registering in the Brokers.

Specified by:

`run` in interface `nSOMAgent`

---

stop

```
public void stop()
```

Performs the stop process of the nSOMTemperatureAgent and the services unregistering in the Brokers.

Specified by:

`stop` in interface `nSOMAgent`

---

unload

```
public void unload()
```

Performs the unload process of the nSOMTemperatureAgent in the Service Execution Platform.

Specified by:

`unload` in interface `nSOMAgent`

Parameters:

`Context` - provided by the nSOMContainer

---

getThermometer

```
public java.lang.String getThermometer()
                                throws java.io.IOException
```

Throws:

`java.io.IOException`

---

getInjuryPrevention

```
public java.lang.String getInjuryPrevention()
                                     throws java.io.IOException
```

Method used to get a lecture of the risk of having a muscular breakdown while performing a workout. This is a composed service made from requests of room temperature, body temperature and heart rate.

Returns:

Risk of having muscular injuries ("HIGH RISK", "MEDIUM RISK", "LOW RISK".

Throws:

`java.io.IOException` - if there is any sort of trouble collecting data from Input/Output classes.

---

getTemperatureControl

```
public java.lang.String getTemperatureControl()
                                    throws java.io.IOException
```

Method used to get a lecture of the temperature levels while performing. a workout. This is a composed service made from requests of room temperature and body temperature.

Returns:

Temperature levels: "VERY LOW", "LOW"", "MEDIUM", "HIGH", "VERY HIGH".

Throws:

`java.io.IOException` - if there is any sort of trouble collecting data from Input/Output classes.

---

wirelessNetworkRequest

```
public java.lang.String
wirelessNetworkRequest(java.lang.String operationName)
                                     throws java.io.IOException
```

Method used to perform requests on the network in order to obtain the required data for the composed services

Parameters:

`operationName` - the name of the operation request to obtain the data for the wider composed service.

Returns:

the value (formatted to a String) of the requested data

Throws:

`java.io.IOException` - if there is any sort of trouble collecting data from Input/Output classes.

---

tomarTemperatura

```
public double tomarTemperatura(java.lang.String serviceInvocationSMD)
```

Method used to extract the temperature value from a JSON answer.

Parameters:

`serviceInvocationSMD` - the JSON answer that is containing the temperature value.

Returns:

the specific temperature value.

---

bluetoothRequest

```
public double bluetoothRequest(java.lang.String serviceInvocationSMD)
```

Method used to extract the value obtained from the Bluetooth-associated mote from a JSON answer.

Parameters:

`serviceInvocationSMD` - the JSON answer that is containing the Bluetooth-associated value.

Returns:

the specific Bluetooth-associated value.

---

receptacle

```
public void receptacle(ServiceContext operationInvocation)
```

Implements the receptacle of the nSOMTemperatureAgent. This method is invoked to pass the service invocations to the agents.

Specified by:

`receptacle` in interface `nSOMAgent`

Parameters:

`ServiceContext` - of the service invocation

---

sendDataRequestToWSN

```
public void sendDataRequestToWSN(java.lang.String serviceInvocationSMD)
```

Process an incoming service request from the DiYSE cloud to the WSN, sending it to the sensor network via Broker node

Parameters:

`serviceInvocationSMD` - invocation in SMD

---

receiveDataRequestFromWSN

```
public java.lang.String receiveDataRequestFromWSN()
                                   throws java.io.IOException
```

Process an incoming service request from the DiYSE cloud to the WSN, waiting the service response from the sensor network via Broker node

Returns:

serviceResponseSMD response in SMD

Throws:

`java.io.IOException`

---

scanAlarms

```
public void scanAlarms()
```

---

getServiceAgentObject

`public nSOMServiceAgentObject getServiceAgentObject()`

Obtain the service description object for this nSOMTemperatureAgent.

Specified by:

`getServiceAgentObject` in interface `nSOMAgent`

Returns:

nSOMServiceAgentObject of the nSOMNoiseAgent

---

getnSOMAgentDescription

`public java.lang.String getnSOMAgentDescription()`

Obtain the nSOMAgentDescription. This method provides the name of the temperature agent which have been invoked.

Specified by:

`getnSOMAgentDescription` in interface `nSOMAgent`

Returns:

String with the agent description

---

nSOMOrchestratorAlarmsAgent.java:

```
public class nSOMOrchestratorAlarmsAgent
extends java.lang.Object
implements nSOMAgent
```

Project: Lifewear

Field Summary	
`static int`	**ID_HBT** Variable used to have a value for maximum body temperature, the actual temperature value will be added to this identifier.
`static int`	**ID_HET** Variable used to have a value for maximum environmental temperature, the actual temperature value will be added to this identifier.
`static int`	**ID_HR** Variable used to have a value for heart rate, the actual heart rate value will be added

	to this identifier.
static int	**ID_LBT** Variable used to have a value for minimum body temperature, the actual temperature value will be added to this identifier.
static int	**ID_LET** Variable used to have a value for minimum environmental temperature, the actual temperature value will be added to this identifier.
static double	**MAX_BODY_TEMPERATURE** Variable used to have a value for maximum body temperature.
static int	**MAX_HEART_RATE** Variable used to have a value for maximum heart rate.
static double	**MAX_ROOM_TEMPERATURE** Variable used to have a value for maximum room temperature.
static double	**MIN_BODY_TEMPERATURE** Variable used to have a value for minimum body temperature.
static int	**MIN_HEART_RATE** Variable used to have a value for minimum heart rate.
static double	**MIN_ROOM_TEMPERATURE** Variable used to have a value for minimum room temperature.

Constructor Summary
nSOMOrchestratorAlarmsAgent() Constructor of the nSOMTemperatureAgent Class.

Method Summary	
double	**bluetoothRequest**(java.lang.String serviceInvocationSMD) Method used to extract the value obtained from the Bluetooth-associated mote from a JSON answer.
int	**checkAlarm**(double envTemp, double bodyTemp, double heartRate) This method is used to check whether the received values are within the

	expected tolerances that the thresholds are defining or there is any value under or over the defined range.
void	**enviarAlarmaESB**(int alarma) Method used to send an alarm to the Enterprise Serial Bus, whenever it is suitable.
void	**enviarTemperaturaMota**(int temp) Method used to send the environmental temperature to the mote that is being carried by the user.
java.lang.String	**getnSOMAgentDescription**() Obtain the nSOMAgentDescription.
nSOMServiceAgentObje ct	**getServiceAgentObject**() Obtain the service description object for this nSOMTemperatureAgent.
int	**isAlarmed**()
void	**load**(nSOMContext nSOMContext) Performs the load process of the nSOMTemperatureAgent in the Service Execution Platform.
java.lang.String	**receiveDataRequestFromWSN**() Process an incoming service request from the DiYSE cloud to the WSN, waiting the service response from the sensor network via Broker node
void	**receptacle**(ServiceContext operationInvocation) Implements the receptacle of the nSOMTemperatureAgent.
double	**requestBodyTemperature**() method used to obtain the body temperature, so as to know if the value is surpassing the low or high level threshold.
double	**requestEnvironmentalTemperature**() method used to obtain the environmental temperature.
int	**requestHeartRate**() method used to obtain the heart rate, so as to know if the value is surpassing the low or high level threshold.
void	**run**() Performs the run process of the nSOMTemperatureAgent and the services registering in the Brokers.

void	`sendDataRequestToWSN(java.lang.String serviceInvocationSMD)` Process an incoming service request from the DiYSE cloud to the WSN, sending it to the sensor network via Broker node
void	`stop()` Performs the stop process of the nSOMTemperatureAgent and the services unregistering in the Brokers.
double	`tomarTemperatura(java.lang.String serviceInvocationSMD)` Method used to extract the temperature value from a JSON answer.
void	`unload()` Performs the unload process of the nSOMTemperatureAgent in the Service Execution Platform.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

MAX_ROOM_TEMPERATURE

```
public static final double MAX_ROOM_TEMPERATURE
```

Variable used to have a value for maximum room temperature.

See Also:

Constant Field Values

MIN_ROOM_TEMPERATURE

```
public static final double MIN_ROOM_TEMPERATURE
```

Variable used to have a value for minimum room temperature.

See Also:

Constant Field Values

MAX_BODY_TEMPERATURE

```
public static final double MAX_BODY_TEMPERATURE
```

Variable used to have a value for maximum body temperature.

See Also:

Constant Field Values

MIN_BODY_TEMPERATURE

```
public static final double MIN_BODY_TEMPERATURE
```

Variable used to have a value for minimum body temperature.

See Also:

Constant Field Values

MAX_HEART_RATE

```
public static final int MAX_HEART_RATE
```

Variable used to have a value for maximum heart rate.

See Also:

Constant Field Values

MIN_HEART_RATE

```
public static final int MIN_HEART_RATE
```

Variable used to have a value for minimum heart rate.

See Also:

Constant Field Values

ID_LET

```
public static final int ID_LET
```

Variable used to have a value for minimum environmental temperature, the actual temperature value will be added to this identifier.

See Also:

Constant Field Values

ID_HET

```
public static final int ID_HET
```

Variable used to have a value for maximum environmental temperature, the actual temperature value will be added to this identifier.

See Also:

Constant Field Values

ID_HR

`public static final int ID_HR`

Variable used to have a value for heart rate, the actual heart rate value will be added to this identifier.

See Also:

Constant Field Values

ID_LBT

`public static final int ID_LBT`

Variable used to have a value for minimum body temperature, the actual temperature value will be added to this identifier.

See Also:

Constant Field Values

ID_HBT

`public static final int ID_HBT`

Variable used to have a value for maximum body temperature, the actual temperature value will be added to this identifier.

See Also:

Constant Field Values

Constructor Detail

nSOMOrchestratorAlarmsAgent

`public nSOMOrchestratorAlarmsAgent()`

Constructor of the nSOMTemperatureAgent Class.

Method Detail

load

`public void load(nSOMContext nSOMContext)`

Performs the load process of the nSOMTemperatureAgent in the Service Execution Platform.

Specified by:

`load` in interface `nSOMAgent`

Parameters:

`Context` - provided by the nSOMContainer

---

run

`public void` **`run`**`()`

Performs the run process of the nSOMTemperatureAgent and the services registering in the Brokers.

Specified by:

`run` in interface `nSOMAgent`

---

stop

`public void` **`stop`**`()`

Performs the stop process of the nSOMTemperatureAgent and the services unregistering in the Brokers.

Specified by:

`stop` in interface `nSOMAgent`

---

unload

`public void` **`unload`**`()`

Performs the unload process of the nSOMTemperatureAgent in the Service Execution Platform.

Specified by:

`unload` in interface `nSOMAgent`

Parameters:

`Context` - provided by the nSOMContainer

---

isAlarmed

`public int` **`isAlarmed`**`()`

---

requestEnvironmentalTemperature

```
public double requestEnvironmentalTemperature()
                                       throws java.io.IOException
```

method used to obtain the environmental temperature. This datum will be sent to mote that is in charge of gathering all the simple data , analyze it and send an alarm to the final user watch if it has to be done.

Returns:

the environmental temperature value.

Throws:

`java.io.IOException` - if there is any sort of trouble collecting data from Input/Output classes.

---

tomarTemperatura

```
public double tomarTemperatura(java.lang.String serviceInvocationSMD)
```

Method used to extract the temperature value from a JSON answer.

Parameters:

`serviceInvocationSMD` - the JSON answer that is containing the temperature value.

Returns:

the specific temperature value.

---

requestBodyTemperature

```
public double requestBodyTemperature()
                                throws java.io.IOException
```

method used to obtain the body temperature, so as to know if the value is surpassing the low or high level threshold.

Returns:

the environmental temperature value.

Throws:

`java.io.IOException` - if there is any sort of trouble collecting data from Input/Output classes.

---

requestHeartRate

```
public int requestHeartRate()
                     throws java.io.IOException
```

method used to obtain the heart rate, so as to know if the value is surpassing the low or high level threshold.

Returns:

the environmental temperature value.

Throws:

`java.io.IOException` - if there is any sort of trouble collecting data from Input/Output classes.

---

bluetoothRequest

```
public double bluetoothRequest(java.lang.String serviceInvocationSMD)
```

Method used to extract the value obtained from the Bluetooth-associated mote from a JSON answer.

Parameters:

`serviceInvocationSMD` - the JSON answer that is containing the Bluetooth-associated value.

Returns:

the specific Bluetooth-associated value.

---

checkAlarm

```
public int checkAlarm(double envTemp,
                      double bodyTemp,
                      double heartRate)
```

This method is used to check whether the received values are within the expected tolerances that the thresholds are defining or there is any value under or over the defined range.

Parameters:

`envTemp` - the received environmental temperature

`bodyTemp` - the received body temperature

`heartRate` - the received heart rate

Returns:

0 if there is no alarm, a three-figured number, composed by the identifier + the received value

---

enviarAlarmaESB

```
public void enviarAlarmaESB(int alarma)
                     throws java.io.IOException
```

Method used to send an alarm to the Enterprise Serial Bus, whenever it is suitable.

Parameters:

`alarma` - a three-figured number containing the kind and the value of the reading that triggered the alarm.

Throws:

`java.io.IOException` - if there is any sort of trouble collecting data from Input/Output classes.

---

enviarTemperaturaMota

```
public void enviarTemperaturaMota(int temp)
                            throws java.io.IOException
```

Method used to send the environmental temperature to the mote that is being carried by the user.

Parameters:

`temp` - the temperature value that is sent to the mote.

Throws:

`java.io.IOException` - if there is any sort of trouble collecting data from Input/Output classes.

---

receptacle

```
public void receptacle(ServiceContext operationInvocation)
```

Implements the receptacle of the nSOMTemperatureAgent. This method is invoked to pass the service invocations to the agents.

Specified by:

`receptacle` in interface `nSOMAgent`

Parameters:

`ServiceContext` - of the service invocation

---

sendDataRequestToWSN

```
public void sendDataRequestToWSN(java.lang.String serviceInvocationSMD)
```

Process an incoming service request from the DiYSE cloud to the WSN, sending it to the sensor network via Broker node

Parameters:

`serviceInvocationSMD` - invocation in SMD

---

receiveDataRequestFromWSN

```
public java.lang.String receiveDataRequestFromWSN()
                                         throws java.io.IOException
```

Process an incoming service request from the DiYSE cloud to the WSN, waiting the service response from the sensor network via Broker node

Returns:

serviceResponseSMD response in SMD

Throws:

`java.io.IOException`

---

getServiceAgentObject

```
public nSOMServiceAgentObject getServiceAgentObject()
```

Obtain the service description object for this nSOMTemperatureAgent.

Specified by:

`getServiceAgentObject` in interface `nSOMAgent`

Returns:

nSOMServiceAgentObject of the nSOMNoiseAgent

---

getnSOMAgentDescription

```
public java.lang.String getnSOMAgentDescription()
```

Obtain the nSOMAgentDescription. This method provides the name of the temperature agent which have been invoked.

Specified by:

`getnSOMAgentDescription` in interface `nSOMAgent`

Returns:

String with the agent description

Classes related to the Temperature agent:

TemperatureSensor.java

```
public class TemperatureSensor
extends java.lang.Object
```

Project: lifewear

Constructor Summary
`TemperatureSensor()` Creates a Temperature sensor instance.

Method Summary

double	getCelsiusTemperature() Obtain a temperature measure in celsius degress.
double	getFahrenheitTemperature() Obtain a temperature measure in fahrenheit degress.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

TemperatureSensor

```
public TemperatureSensor()
```

Creates a Temperature sensor instance.

Method Detail

getCelsiusTemperature

```
public double getCelsiusTemperature()
                                   throws java.io.IOException
```

Obtain a temperature measure in celsius degress.

Returns:

Celsius temperature measure (double)

Throws:

`java.io.IOException`

---

getFahrenheitTemperature

```
public double getFahrenheitTemperature()
                                      throws java.io.IOException
```

Obtain a temperature measure in fahrenheit degress.

Returns:

Fahrenheit temperature measure (double)

Throws:

`java.io.IOException`

nSOMTemperatureAgent1.java

```
public class nSOMTemperatureAgent1
extends java.lang.Object
```

Project: lifewear

Constructor Summary
`nSOMTemperatureAgent1()` Constructor of the nSOMTemperatureAgent Class.

Method Summary	
`java.lang.String`	`getnSOMAgentDescription()` Obtain the nSOMAgentDescription.
`nSOMServiceAgentObject`	`getServiceAgentObject()` Obtain the service description object for this nSOMTemperatureAgent.
`double`	`getTemperature1(byte unit, int resolution)` Obtain a temperature measure, in double format.
`void`	`load(nSOMContext nSOMContext)` Performs the load process of the nSOMTemperatureAgent in the Service Execution Platform.
`void`	`receptacle(ServiceContext operationInvocation)` Implements the receptacle of the nSOMTemperatureAgent.
`void`	`run()` Performs the run process of the nSOMTemperatureAgent and the services registering in the Brokers.
`void`	`stop()` Performs the stop process of the nSOMTemperatureAgent and the services unregistering in the Brokers.
`void`	`unload()` Performs the unload process of the nSOMTemperatureAgent in the Service Execution Platform.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

nSOMTemperatureAgent1

`public` **`nSOMTemperatureAgent1`**`()`

Constructor of the nSOMTemperatureAgent Class.

Method Detail

load

`public void` **`load`**`(nSOMContext nSOMContext)`

Performs the load process of the nSOMTemperatureAgent in the Service Execution Platform.

Parameters:

`Context` - provided by the nSOMContainer

---

run

`public void` **`run`**`()`

Performs the run process of the nSOMTemperatureAgent and the services registering in the Brokers.

---

stop

`public void` **`stop`**`()`

Performs the stop process of the nSOMTemperatureAgent and the services unregistering in the Brokers.

---

unload

`public void` **`unload`**`()`

Performs the unload process of the nSOMTemperatureAgent in the Service Execution Platform.

Parameters:

`Context` - provided by the nSOMContainer

---

getTemperature1

```
public double getTemperature1(byte unit,
                              int resolution)
```

Obtain a temperature measure, in double format.

Parameters:

`unit` - of the measure (C, F, K).

`resolution` - of the double format measure (0, 1, 2).

Returns:

Noise measure (double primitive data type)

---

receptacle

```
public void receptacle(ServiceContext operationInvocation)
```

Implements the receptacle of the nSOMTemperatureAgent. This method is invoked to pass the service invocations to the agents.

Parameters:

`ServiceContext` - of the service invocation

---

getServiceAgentObject

```
public nSOMServiceAgentObject getServiceAgentObject()
```

Obtain the service description object for this nSOMTemperatureAgent.

Returns:

nSOMServiceAgentObject of the nSOMNoiseAgent

---

getnSOMAgentDescription

```
public java.lang.String getnSOMAgentDescription()
```

Obtain the nSOMAgentDescription. This method provides the name of the temperature agent which have been invoked.

Returns:

String with the agent description

Finally, it is also offered the runBundle.java class that is executed on the PC that has installed the ESB in order to initialize the base station:

runBundle.java

```
public class runBundle
extends java.lang.Object
```

Project: Lifewear

Field Summary	
static int	LIMITELEDS

Constructor Summary
runBundle()

Method Summary	
java.lang.String	generateSMDRequestFromCloud(java.lang.String receivedServiceInvocation) Generate the SMD invocation from the cloud invocation
java.lang.String	getTemperature(java.lang.String id) Method used to get the environmental temperature after a user request.
java.lang.String	getThermometre() Method used to get temperature readings and show them as LEDs in different colors.
static void	main(java.lang.String[] args) Start up the host application.
void	receiveDataRequestFromCloud(java.lang.String receivedRequest) Receive service request from the DiYSE cloud to the WSN
java.lang.String	receiveDataRequestFromWSN() Process an incoming service request from the DiYSE cloud to the WSN, waiting the service response from the sensor network via Broker node

void	`sendDataRequestToWSN(java.lang.String serviceInvocationSMD)` Process an incoming service request from the DiYSE cloud to the WSN, sending it to the sensor network via Broker node
void	`sendDataResponseToCloud(java.lang.String receivedResponseSMD)` Send received data to the DiYSE cloud from the WSN
double	`tomarTemperatura(java.lang.String serviceInvocationSMD)`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

LIMITELEDS

```
public static final int LIMITELEDS
```

See Also:

Constant Field Values

Constructor Detail

runBundle

```
public runBundle()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
                 throws java.lang.Exception
```

Start up the host application.

Parameters:

`args` - any command line arguments

Throws:

`java.lang.Exception`

getTemperature

```
public java.lang.String getTemperature(java.lang.String id)
                                throws java.rmi.RemoteException
```

Method used to get the environmental temperature after a user request.

Parameters:

`id` - identifier of the mote the temperature is being asked from

Returns:

the temperature value converted to a String.

Throws:

`java.rmi.RemoteException` - in case the RMI part has any issue.

---

getThermometre

```
public java.lang.String getThermometre()
                                throws java.rmi.RemoteException
```

Method used to get temperature readings and show them as LEDs in different colors. It has only testing purposes.

Returns:

message of requested thermometer invocation

Throws:

`java.rmi.RemoteException` - in case the RMI part has any issue.

---

sendDataResponseToCloud

```
public void sendDataResponseToCloud(java.lang.String receivedResponseSMD)
```

Send received data to the DiYSE cloud from the WSN

Parameters:

`receivedResponse` - data received from the wireless sensor network

---

receiveDataRequestFromCloud

```
public void receiveDataRequestFromCloud(java.lang.String receivedRequest)
```

Receive service request from the DiYSE cloud to the WSN

Parameters:

`receivedRequest` - request received from the cloud

---

sendDataRequestToWSN

```
public void sendDataRequestToWSN(java.lang.String serviceInvocationSMD)
```

Process an incoming service request from the DiYSE cloud to the WSN, sending it to the sensor network via Broker node

Parameters:

`serviceInvocationSMD` - invocation in SMD

---

receiveDataRequestFromWSN

```
public java.lang.String receiveDataRequestFromWSN()
                                          throws java.io.IOException
```

Process an incoming service request from the DiYSE cloud to the WSN, waiting the service response from the sensor network via Broker node

Returns:

serviceResponseSMD response in SMD

Throws:

`java.io.IOException`

---

generateSMDRequestFromCloud

```
public java.lang.String
generateSMDRequestFromCloud(java.lang.String receivedServiceInvocation)
```

Generate the SMD invocation from the cloud invocation

Parameters:

`receivedServiceInvocation` - invocation from the cloud

Returns:

SMD description of the data request invocation

---

tomarTemperatura

```
public double tomarTemperatura(java.lang.String serviceInvocationSMD)
```

Annex II

Chart of available motes

Sensor Node Name	Microcontroller	Transceiver	Program + Data Memory	External Memory	Programming	Remarks
Arago SystemsWiSMote Dev	MSP430F5437	CC2520	RAM : 16 Kbytes Flash : 256 Kbytes	up to 8 Mbits	C	Contiki and 6LoWPan supported.
Arago SystemsWiSMote Mini	ATMEGA128 RFA2	ATMEGA128 RFA2	RAM : 16 Kbytes Flash : 128 Kbytes E²PROM : 4Kbytes		C	Contiki and 6LoWPan supported.
AVRraven Atmel AVR#Raven wireless kit	AtMega1284p + ATmega3290p	AT86RF230	128 Kbytes + 16 Kbytes	256 kB?	C	
COOKIES	ADUC841, MSP430	ETRX2 TELEGESIS, ZigBit 868/915	4 Kbytes + 62 Kbytes	4 Mbit	C	Platform with hardware reconfigurability (Spartan 3FPGA based or Actel Igloo)
BEAN	MSP430F169	CC1000 (300-1000 MHz) with 78.6 kbit/s		4 Mbit		YATOS Support.
BTnode	Atmel ATmega 128L (8 MHz	Chipcon CC1000 (433-915 MHz) and Bluetooth	64+180 K RAM	128K FLASH	C and nesC Programming	BTnut and TinyOS support.

	@ 8 MIPS)	(2.4 GHz)		ROM, 4K EEPROM		
COTS	ATMEL Microcontroller 916 MHz					
Dot	ATMEGA163		1K RAM	8-16K Flash	weC	
EPIC mote	Texas Instruments MSP430 microcontroller	250 kbit/s 2.4 GHz IEEE 802.15.4 Chipcon Wireless Transceiver	10k RAM	48k Flash		TinyOS.
Egs [1]	ARM Cortex M3	CC2520, Mitsumi's class 2 Bluetooth module		2 Gbit		TinyOS.
Eyes	MSP430F149	TR1001		8 Mbit		PeerOS Support.
EyesIFX v1	MSP430F149	TDA5250 (868 MHz) FSK		8 Mbit		TinyOS Support.
EyesIFX v2	MSP430F1611	TDA5250 (868 MHz) FSK		8 Mbit		TinyOS Support.
FlatMesh FM1	16 MHz	802.15.4-compliant		660 sensor readings	Over-air control	Commercial system, for digital sensors.
FlatMesh FM2	16 MHz	802.15.4-compliant		660 sensor readings	Over-air control	Commercial system, built-in tilt sensor.
GWnode	PIC18LF8722	BiM (173 MHz) FSK	64k RAM	128k flash	C	Custom OS.

IMote	ARM core 12 MHz	Bluetooth with the range of 30 m	64K SRAM	512K Flash		TinyOS Support.
IMote 1.0	ARM 7TDMI 12-48 MHz	Bluetooth with the range of 30 m	64K SRAM	512K Flash		TinyOS Support.
IMote 2.0	Marvell PXA271 ARM 11-400 MHz	TI CC2420 802.15.4/ZigBee compliant radio	32 MB SRAM	32 MB Flash		Microsoft .NET Micro, Linux, TinyOS Support.
INDriya_CS_03A14[2]	Atmel ATmega 128L [3]	IEEE 802.15.4 compliant XBee radios	128 KB FLASH + 4 KB RAM	Expansion available	C-programming & nesC compliant	Comprehensive Sensor mote with: Ambient Light,temperature,accelerometer,JPEG camera,PIR, sound sensor, TinyOS TinyOS compliant, IPv6 network supportive stacks for internetworking & so on.
Iris Mote	ATmega 1281	Atmel AT86RF230 802.15.4/ZigBee compliant radio	8K RAM	128K Flash	nesC	Mote Runner, TinyOS, MoteWorks Support.
KMote	TI MSP430	250 kbit/s 2.4 GHz IEEE 802.15.4 Chipcon Wireless Transceiver	10k RAM	48k Flash		TinyOS and SOS Support.
Mica	ATmega 1034 MHz 8-bit CPU	RFM TR1000 radio 50 kbit/s	128+4K RAM	512K Flash	nesC Programming	TinyOS Support.
Mica2	ATMEGA 128L	Chipcon 868/916 MHz	4K RAM	128K Flash		TinyOS, SOS and MantisOS Support.
Mica2Dot	ATMEGA 128		4K RAM	128K		

				Flash		
MicaZ	ATMEGA 128	TI CC2420 802.15.4/ZigBee compliant radio	4K RAM	128K Flash	nesC	TinyOS, SOS, MantisOS and Nano-RK Support.
Monnit WIT	TI CC1110	868/900 MHz	4K RAM		C#	multi sensor boards.
Mulle	Renesas M16C	Atmel AT86RF230 802.15.4 / Bluetooth 2.0	31K RAM	384K+4K Flash, 2 MB EEPROM	nesC, C programming	Contiki, TinyOS, lwIP: TCP/IP and Bluetooth Profiles: LAP, DUN, PAN and SPP Support.
NeoMote	ATmega 128L	TI CC2420 802.15.4/ZigBee compliant radio	4K RAM	128K Flash	nesC	TinyOS, SOS, MantisOS, Nano-RK and Xmesh Support, Industrial end-use product.
Nymph	ATMEGA128L	CC1000		64 kB EEPROM		MantisOS Support.
PowWow	MSP430F1612	TI CC2420 802.15.4/ZigBee compliant radio	55kB Flash + 5kB RAM		C programming : MSPGCC, IAR	Open source; open hardware; research platform.
Preon32	ARM Cortex M3	Atmel AT86RF231 (2.4 GHz)	64kB RAM + 256kB Flash	8 Mbit	Java	virtual machine, Contiki and 6LoWPan supported.
Redbee	MC13224V	2.4 GHz 802.15.4	96 KB RAM + 120 KB Flash		GCC (seemc1322x.devl.org), IAR	Contiki; standalone.
Rene	ATMEL8535	916 MHz radio with bandwidth of 10 kbit/s	512 bytes RAM	8K Flash		TinyOS Support.

SenseNode	MSP430F1611	Chipcon CC2420	10K RAM	48K Flash	C and NesC programming	GenOS and TinyOS Support.
Shimmer	MSP430F1611	802.15.4 Shimmer SR7 (TI CC2420)	48 KB Flash 10 KB RAM	2 GB microSD Card	nes C and C Programming	TinyOS Support. Built in 3 Axis Accel, Tilt/Vib Sensor. Full range of expansion modules.
Sun SPOT	ARM 920T	802.15.4	512K RAM	4 MB Flash	Java	Squawk Java ME Virtual Machine.
Telos	MSP430		2K RAM			
TelosB	Texas Instruments MSP430 microcontroller	250 kbit/s 2.4 GHz IEEE 802.15.4 Chipcon Wireless Transceiver	10k RAM	48k Flash		Contiki, TinyOS, SOS and MantisOS Support.
Tinynode	Texas Instruments MSP430 microcontroller	Semtech SX1211	8K RAM	512K Flash	C Programming	TinyOS.
T-Mote Sky	Texas Instruments MSP430 microcontroller	250 kbit/s 2.4 GHz IEEE 802.15.4 Chipcon Wireless Transceiver	10k RAM	48k Flash		Contiki, TinyOS, SOS and MantisOS Support.
Waspmote	Atmel ATmega 1281	ZigBee/802.15.4/DigiMesh/RF, 2.4 GHz/868/900 MHz	8K SRAM	128K FLASH ROM, 4K EEPROM , 2 GB SD card	C/Processing	GPRS, Bluetooth, GPS modules, sensor boards.

weC	Atmel AVR AT90S2313	RFM TR1000 RF				
Wireless RS485	Atmega 128L	Chipcon CC2420 + Amplifier 250 kbit/s 2.4 GHz IEEE 802.15.4	4k RAM	128k Flash		Xmesh, TinyOS.
XYZ	ML67 series ARM/THUMB microcontroller	CC2420 Zigbee compliant radio from Chipcon	32K RAM	256K Flash	C Programming	SOS Operating System Support.
XM1000	TI's MSP430F2618	TI's CC2420	8K RAM	116K FLASH ROM		Contiki, TinyOS, SOS and MantisOS Support.
Zolertia Z1	Texas Instruments MSP430F2617	Chipcon CC2420 2.4 GHz IEEE 802.15.4 Wireless Transceiver	8 KB RAM	92 KB Flash	C, nesC	Contiki and TinyOS Support. 16 Mbit external flash + 2 digital on-board sensors.
FireFly	Atmel ATmega 1281	Chipcon CC2420	8K RAM	128K FLASH ROM, 4K EEPROM	C Programming	Nano-RK RTOS Support.
Ubimote1	TI's CC2430 SOC based on 8051 Core	TI's CC2430	8K RAM	128K FLASH ROM	C Programming	TI's ZStack, TinyOS Support.
Ubimote2	TI's MSP430F2618	TI's CC2520	8K RAM	116K FLASH ROM	C Programming	TI's ZStack Support.
VEmesh	TI MSP430	Semtech SX1211/1231, TI TRF6903	512B RA M	8K FLAS H	Over-the-air Programming	FHSS; Interface to MODBUS, DALI, RS-232/485, TCP/IP.

panStamp	Atmega328P	TI CC1101 (868/915 MHz)	2K RAM	32K FLASH, 1K EEPROM	C/C++	Software compatible with Arduino

Chart 43: list of available wireless sensor nodes [Nodes 12].

Another chart of gateway sensor nodes is as follows:

	Microcontroller	**Transceiver**	**Interface (USB/Serial/Wifi/Ethernet)**	**Program memory**	**External Memory**
ADVANTICSYS SG1000	Dual Core Atom D525 2x1.8Ghz	802.15.4-compliant	RS232,USB,TCP/IP		2GB RAM, 160GB HDD
DWARA_CS_03A30	Atmel ATmega 128L [4]	WPAN:IEEE 802.15.4 compliant XBee radio GSM/GPRS Modem:SIM300S	UART based serial connection to SIM300S	128 KB Flash	---
Shimmer Span	MSP430F1611	802.15.4 Shimmer SR7 (TI CC2420)	USB Flashdrive - Doesn't block adjacent USB ports	10 KB RAM 48 KB Flash	
Stargate	IntelPXA255	802.11	Serial connection to WSN	64 MB SDRAM	32 MB Flash
FlatMesh FMG-S	16 MHz	802.15.4-compliant	Serial connection to FlatMesh FM1, FM2		660 sensor readings

VEmesh	TI MSP430	Semtech SX1211/1231, TI TRF6903	MODBUS, USB, RS-232/485, TCP/IP	2K RAM + 16K FLASH	Expansion available

Chart 44: list of available gateway sensor nodes [Nodes 12].

Bibliographic references

[24X7 07] Website of the 24 X 7 magazine, January 2007, http://www.24x7mag.com/issues/articles/2007-01_12.asp.

[Aimglobal 12] Website of the Association for automatic Identification and Mobility, August 2012, http://www.aimglobal.org/technologies/RFID/what_is_rfid.asp.

[Aura 02] Website of the Aura project, December 2002, http://www.cs.cmu.edu/~aura/.

[Becker&04] Christian Becker, Marcus Handte, Gregor Schiele, Kurt Rothermel. "*PCOM- A Component System for Pervasive Computing*". Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. PerCom 2004. 14-17 March 2004.

[Berkeley 03] Website of the Department of Electrical Engineering and Computer Sciences University of California, Berkeley, July 1999, http://www.cs.berkeley.edu/Weiser/bio.shtml.

[BioHarness™ 3 10] Zephyr "*Zephyr BioHarness™ 3 Product description*" datasheet, November 2010. Zephyr Technology, Annapolis, USA.

[Bluegiga 12] Website of Sparkfun Electronics, August 2012,

https://www.sparkfun.com/products/8952.

[BTNode 09] Website of the BTNodes created by ETH, April 2009,

http://www.btnode.ethz.ch/static_docs/doxygen/btnut/.

[Campo 04] María Teresa Campo Vázquez. "*Tecnologías middleware para el desarrollo de servicios en entornos de computación ubicua".* Phd. Thesis, Universidad Carlos III de Madrid. May 2004.

[Castells 96] "*The rise of the Network Society*". Blackwell Publishers, ISBN 1557866163. 1996.

[Contiki 12] Website of Contiki operating system, August 2012, http://www.contiki-os.org/#about

[CORDIS 08] Website of CORDIS portal, December 2008, http://cordis.europa.eu/search/index.cfm?fuseaction=news.document&N_RCN=30283.

[Cricket 12] "*MEMSIC Cricket datasheet*". MEMSIC corporation. Andover, USA. 2010.

[Cuerva 12] Alexandra Cuerva García. "*Redes y Servicios Ubicuos en Dispositivos llevables para el Internet de las Cosas*". Final Degree Dissertation, Universidad Politécnica de Madrid, September 2012.

[Davies&08] John Francis Davies (Editor), Marko Grobelnik (Editor), Dunja Mladenic (Editor). "*Semantic Knowledge Management: Integrating Ontology Management, Knowledge Discovery, and Human Language*

Technologies". Springer, ISBN 978-3-540-88844-4. 2008-.

[DCOM 12] Website of Microsoft, August 2012,

http://msdn.microsoft.com/library/cc201989.aspx.

[de Castro 11] Antonio martínez de Castro. "*Sistema de ayuda al bombero en incendios forestales*", Final Degree Dissertation, Universidad Pontificia Comillas. September 2011.

[Donohoo 11] Donohoo, B.K. "*AURA: An Application and User Interaction Aware Middleware Framework for Energy Optimization in Mobile Devices*". 29th International Conference on Computer Design, 2011 IEEE. 9-12 October 2011.

[Emmerich&02] Capra, L., Mascolo, C., Zachariadis, S., Emmerich, W. "*Towards a mobile computing middleware: Synergy of reflection and mobile code techniques*". The Eighth IEEE Workshop on Future Trends of Distributed Computing Systems, 2001. FTDCS 2001. 31 Oct.-2 Nov. 2001.

[English 11] Website of the English language-based portal, February 2011, http://english.stackexchange.com/questions/12891/whats-a-tuple-in-normal-english.

[ETH 08] Website of the BTNode motes, manufactured by ETH, 2007, http://www.btnode.ethz.ch/.

[ETH 08b] "BTnode rev3 – Product Brief". ETH Insitute. Zürich, Switzerland, http://www.btnode.ethz.ch/pub/uploads/Main/btnode_rev3.24_productbrief.pdf. March 2006.

[Familiar 09] Miguel Santos Familiar Cabrero. "*Sistema de Gestión de Suscripción para Redes Inalámbricas de Sensores*". Final Degree Dissertation, Universidad Politécnica de Madrid, February 2009.

[García-Hernando&08] Ana-Belén García-Hernando, José-Fernán Martínez-Ortega, Juan-Manuel López-Navarro, Aggeliki Prayati and Luis Redondo-López. "*Problem Solving for Wireless Sensor Networks*". Springer, ISBN 978-1-84800-202-9. 2008.

[González 10] Website of a project related with Wireless Sensor Networks made by José Gonzalez and César Reyes, December 2010, http://profesores.elo.utfsm.cl/~agv/elo323/2s10/projects/ReyesGonzalez/dispositivos.html.

[Greenfield 06] Adam Greenfield. "*Everyware: The dawning age of ubiquitous computing*". New Riders, ISBN 0-321-38401-6. 2006. Pages 144-147.

[Grimm 04] Robert Grimm. "One.world: Experiences with a Pervasive Computing Architecture", http://cs1.cs.nyu.edu/rgrimm/one.world/papers/pervasive04.pdf. 2004.

[GRyS&12] Pedro Castillejo, Alejandra Cuerva, Huang Yuanjiang, Jesús Rodríguez.

"*Architectural overview of Lifewear*". Slide presentation, March 2012.

[Hadim&06] Salem Hadim, Nader Mohamed. "*Middleware Challenges and Approaches for Wireless Sensor Networks*". IEEE Distributed Systems Online, Vol. 7, No. 3. March 2006.

[Haller 09] Stephan Haller. "*Internet of Things: An Integral Part of the Future Internet*". Slide presentation, 13th May 2009.

[Harihar&05] Karthik Harihar, Stan Kurkovsky. "Using Jini to Enable Pervasive Computing Environments". ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference - Volume 1, http://dl.acm.org/results.cfm?h=1&cfid=149408443&cftoken=40775745. March 2005.

[Hewlett-Packard 12] Hewlett-Packard. "Getting Started Guide", August 2012, http://h10032.www1.hp.com/ctg/Manual/bpo50005.pdf.

[Honeywell HX2] Website of the Honeywell device HX2, http://www.honeywellaidc.com/en-US/Pages/product.aspx?category=Wearable&cat=HSM&pid=HX2 . August 2012.

[Hung&04] N.Q, hung, N.C. Ngoc, L.X, hung, Shu Lei, S.Y. Lee. "A Survey on Middleware for Context-Awareness in Ubiquitous Computing Environments", http://www-nishio.ist.osaka-u.ac.jp/~leishu/pub/Survey_Context.pdf .2004.

[Ibrahim 09] N. Ibrahim. "*Orthogonal Classification of Middleware Technologies*". Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, 2009. UBICOMM '09. 11-16 Oct. 2009.

[Intel 11] Website of Intel Software Network, November 2011, http://software.intel.com/en-us/articles/upnp-technology-and-rich-media-for-the-digital-home-part-1/.

[Iris 12] MEMSIC Iris datasheet. MEMSIC Corporation. Andover, USA. August 2012.

[Ironick 05] Website of the web portal Ironick, July 2005, http://ironick.typepad.com/ironick/2005/07/update_on_the_o.html.

[ISGTW 07] Website of the online journal International Science Grid This Week, November 2007, http://www.isgtw.org/feature/isgtw-opinion-anticipating-futures-engineering-expectations-ubiquitous-computing.

[JVM 12] Image from a website devoted to Sun SPOT motes, last access August 2012, http://en.wikipedia.org/wiki/File:Standard_Java_VM_vs_Squawk_Java_VM .svg.

[Kindberg&01] John Barton, Tim Kindberg. "The Challenges and Opportunities of Integrating the Physical World and Networked Systems", January 2001, http://www.hpl.hp.com/techreports/2001/HPL-2001-18.pdf .

[Kindberg 12] Website of the Cooltown project, July 2012, http://champignon.net/cooltown.php.

[Kjaer 07] Kristian Ellebæk Kjær. "*A survey of context-aware middleware*", March 2005, http://www.hydramiddleware.eu/hydra_papers/A_Survey_of_Context-aware_Middleware.pdf.

[Libelium 12] Website of Libelium company, August 2012, http://www.libelium.com/top_50_iot_sensor_applications_ranking/ .

[Libelium 12b] Libelium Waspmote data sheet,

http://www.libelium.com/documentation/waspmote/waspmote-datasheet_esp.pdf. Libelium Comunicaciones. Zaragoza, Spain. August 2012.

[Libelium 12c] Website of Libelium company, August 2012,

http://www.libelium.com/company/about.

[Lifewear 10] Preliminar website of the Lifewear project -2010-,

https://www.euitt.upm.es/uploaded/eventos/Actualidad-Prensa/nota-web-Lifewear.pdf.

[Lifewear 12] Lifewear Deliverable 2.1, "*Platform Requirements Specification*". Pages 24-28.

[Lifewear 12b] Website of the Lifewear project, August 2012, http://www.lifewear.es.

[Lotus 12] MEMSIC Lotus datasheet. MEMSIC Corporation. Andover, USA. August 2012.

[Lübeck 12] Website of a ubiquitous device, last access August 2012, http://www.iti.uni-luebeck.de/?id=112.

[Mann 98] Steve Mann's Keynote Address entitled "Wearable Computing as means for personal empowerment", May 1998, http://wearcam.org/wearcompdef.html.

[MEMSIC 10] Website of MEMSIC company, last access: August 2012, http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html.

[MICA2 12] MEMSIC MICA2 Lotus datasheet. MEMSIC Corporation. Andover, USA. 2012.

[MICA2Dot 12] MEMSIC MICA2Dot Lotus datasheet. MEMSIC Corporation. Andover, USA. 2012.

[MICAz 12] MEMSIC MICAz Lotus datasheet. MEMSIC Corporation. Andover, USA. 2012.

[Microsoft 12]	Website of Microsoft, August 2012, http://msdn.microsoft.com/es-es/library/ms731082.aspx.
[Moog-Crossbow 12]	Web page on Moog-Crossbow site, August 2012, http://www.xbow.com/corporate/about/Timeline.html.
[Nodes 12]	Website of the mote charts, July 2012, http://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes
[Northeastern 10]	Website of the Northeastern University, February 2010, http://www.northeastern.edu/news/stories/2010/02/baseball_shirt.html.
[Obitko 12]	Marek Obitko website, last access: August 2012, http://obitko.com/tutorials/ontologies-semantic-web/semantic-web-architecture.html.
[Oracle 12]	Website of Oracle company, August 2012, http://www.oracle.com/es/index.html.
[Österlind 09]	Fredrik Österlind. "*Contiki Hands-on*" Silde presentation, July 2009, http://es.scribd.com/doc/38040678/16/Contiki-two-communication-stacks.
[Ponnekanti&03]	Ponnekanti, S.R., Johanson, B., Kiciman, E., Fox, A. "*Portability, Extensibility and Robustness in iROS*". Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003. (PerCom 2003). 26 March 2003.
[Radio-electronics 12]	Website of the portal for electronic engineers Radio-communications, August 2012, http://www.radio-electronics.com/info/wireless/ieee-802-15-4/wireless-standard-technology.php.
[Roma&00]	Roman, Manuel, Mickunas Dennis, Kon Fabio, and Campbell Roy H. "*LegORB and Ubiquitous CORBA*". Proceedings of the IFIP/ACM Middleware'2000 Workshop on Reflective Middleware. April 2000.
[Rosslin&11]	Rosslin John Robles, Kyung Jung Kim. "*Securing Child Information Access Control in Ubiquitous Healthcare Systems*", February 2011, http://www.sersc.org/journals/JSE/vol8_no1_2011/4.pdf.
[Rudolph 01]	Website of the Oxygen project, last access August 2012, http://www.oxygen.lcs.mit.edu/.
[Rudolph 01b]	Larry Rudolph. "*Project Oxygen: Pervasive, Human-Centric Computing – An Initial Experience*". Advanced Information Systems Engineering, 13th International Conference (CAiSE2001). June 2001.
[SAI 12]	Web page of SAI Wireless company, last access August 2012, http://www.saiwireless.com/.
[Saif&01]	Saif, U., Greaves, D.J. "*Communication Primitives for Ubiquitous Systems or RPC Considered Harmful*". International Conference on Distributed

	Computing Systems Workshop, 2001. 16-19 April 2001-.
[Seah&09]	Seah, W.K.G. Zhi Ang Eu, Hwee-Pink Tan. "*Wireless Sensor Networks Powered by Ambient Energy Harvesting (WSN-HEAP) – Survey and Challenges*". 1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology, 2009. Wireless VITAE 2009. 17-20 May 2009.
[Sentilla 12]	Website of Sentilla Company, last Access August 2012, www.sentilla.com.
[Sharmin&06]	Sharmin, M.; Ahmed, S.; Ahamed, S.I. "*MARKS (Middleware Adaptability for Resource Discovery, Knowledge Usability and Self-healing) for Mobile Devices of Pervasive Computing Environments*". Third Conference on Information Technology: New Generations, 2006. ITNG 2006. 10-12 April 2006.
[SICS 12]	Website of the Swedish Institute of Computer Science, last access August 2012, http://www.sics.se/
[Sikos 11]	Website of Dr. Leslie Sikos, last access August 2012, http://www.lesliesikos.com/tutorials/rdf/.
[SmartSantander 12]	Website of the SmartSantander deployment, last access August 2012, http://www.smartsantander.eu/.
[SRI 08]	"*Disruptive Technologies APPENDIX F Background: The Internet of Things Global Trends 2025*", September 2008, http://robertoigarza.files.wordpress.com/2008/09/rep-six-technologies-with-potential-impacts-on-us-anexothe-internet-of-things-nic-2008.pdf
[Sun SPOT 12]	Oracle Sun SPOT datasheet. Oracle Corporation. Redwood Shores, USA. August 2012.
[Sun SPOT 12b]	Website for Sun SPOT developers, August 2012, http://www.sunspotworld.com/docs/general-faq.html.
[Tecnalia 12]	Website of Tecnalia company -2012- http://www.tecnalia.com/.
[TelosB 12]	MEMSIC TelosB datasheet. MEMSIC Corporation. Andover, USA. August 2012.
[Teo&10]	Teo, L, Teh, D, Corbitt, B. "*Service Oriented Architecture (SOA): Implications for Australian University Information Systems Curriculum*". Pacific Asia Conference on Information Systems (PACIS). July 9-12, 2010.
[Txarramendieta 06]	Santi Txarramendieta. "*Middleware en computación ubicua: Project Aura y Gaia*". Slide presentation, June 2006.
[UPnP 06]	UPnP Implementers Corporation´s "*UPnP™ Technology – The Simple, Seamless Home Network, UPnP implementers Corporation*", last access August 2012, http://upnp.org/resources/whitepapers/UPnP%20Technology_The%20Sim

ple,%20Seamless%20Home%20Network_whitepaper.pdf.

[Vicente 10] Patricia Vicente Lezaun. "*Virtualización en Redes Inalámbricas de Sensores*" Final Degree Dissertation, Universidad Politécnica de Madrid, September 2010.

[W3C 12] Website of W3C. Frequently Asked Questions, last access August 2012, http://www.w3.org/RDF/FAQ/.

[W3C 12b] Website of W3C. Ontologies, last access August 2012, http://www.w3.org/2001/sw/SW-FAQ#whmustont.

[W3C 12c] Website of W3C. OWL, last access August 2012, http://www.w3.org/TR/owl2-overview/.

[W3C 12d] Website of W3C. Frequently Asked Questions about RIF, last access August 2012, http://www.w3.org/2005/rules/wiki/RIF_FAQ.

[W3C 12e] Website of W3C. RIF, last access August 2012, http://www.w3.org/2005/rules/wiki/RIF_FAQ#What_is_RIF.3F.

[W3C 12f] Website of W3C. SPARQL, last access August 2012, http://www.w3.org/blog/SW/2008/01/15/sparql_is_a_recommendation/.

[Wei 11] Wei Liu. "*A survey on context awareness*". International Conference on Computer Science and Service System (CSSS), 2011. 27-29 June 2011.

[Weiderman&97] Nelson H. Weiderman, John K. Bergey, Dennis B. Smith, Scott R. Tilley. "*Approaches to Legacy System Evolution*", December 1997, http://www.sei.cmu.edu/reports/97tr014.pdf.

[WIMM 12] Website of WIMM manufacturer, last access August 2012, https://my.wimm.com/.

[Yanko 10] Website dedicated to electronic gadgets and appliances, May 2010, http://www.yankodesign.com/2010/05/25/in-2020-we-can-wear-sony-computers-on-our-wrist/.

[Yao&05] Jianchu Yao, Ph.D. and Steve Warren, Ph.D. "*Applying the ISO/IEEE 11073 standards to wearable home health monitoring systems*", Journal of Clinical Monitoring and Computing Vol 19 No 6 2005, Springer, 2006.

[Yastrebova&07] N. Yastrebova, K. Hinzer, D. Masson, S. Desgreniers, H. Schriemer, B. J. Riel, S. Fafard, T. J. Hall. Slide presentation, http://ptlab.site.uottawa.ca/downloads/YastrebovaN_solar_powered_WSN.pdf. 3rd May 2007.

[Yau&03] Yau, Stephen S., Karim Fariaz, "*A Lightweight Middleware Protocol for Ad Hoc Distributed Object Computing in Ubiquitous Computing Environments*". Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003.14-16 May 2003.