View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório Aberto da Universidade do Porto

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards a Self-Managed Framework for Orchestration and Integration of Devices in AAL

João Quarteu Alves

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Hugo José Sereno Lopes Ferreira (Ph.D)

Co-supervisor: Tiago Boldt Pereira de Sousa (M.Sc)

February 13, 2014

Towards a Self-Managed Framework for Orchestration and Integration of Devices in AAL

João Quarteu Alves

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. João Pascoal Faria (Ph.D)

External Examiner: Prof. Ângelo Martins (Ph.D)

Supervisor: Prof. Hugo Sereno Ferreira (Ph.D)

February 13, 2014

Abstract

The *Internet of Things* (IOT) paradigm along with the use of Wireless Sensor Networks (WSNs) is revolutionizing *smart environments*, enabling a new world of products and services that is enhancing people's daily activities. From domotics to surveillance or Ambient Assisted Living(AAL) systems based on WSNs are proliferating worldwide, bringing new challenges to their architecture.

With the continuous aging of the world population, AAL systems can provide continuous monitoring of individuals health status, reducing the constant need to attend health facilities for observation. AAL4ALL is a nation-funded project, involving more than 30 institutes and companies, that aims to build an open ecosystem for AAL. The need to have a framework that allows to have third-party hardware and software on the top of a WSN arose from the fact that any partner could develop its own products and services, once certified.

In the previously described scenario, multiple sensors can be deployed across the household or health institutes to monitor one or more users. Powered by short-range communication technologies, these devices require sensor networks to seamlessly connect to, wherever the user might be, keeping the patient always in range of one of the nodes in the sensor network to ensure ubiquitous monitoring.

This dissertation proposes an architecture for a generic, scalable, fault-tolerant and auto-configurable sensor network, supporting Bluetooth, enabling dynamic loading of drivers developed by third-party in order to allow a continuous evolution of the system without the need for manual configuration.

The proposed solution is based in a *Publisher/Subscriber* pattern, where sensors are continuously publishing information to a message queue in the WSN. Subscribers could be either actuators and local or *cloud* services, that are consuming the information gathered. In order to achieve concurrency and fault-tolerance, the *Actor model* pattern is adopted. Since low-computing devices are used, the fact of having multiple nodes distributed across the building was exploited to build a peer-to-peer and fault-tolerant cluster.

To assess the validity of this dissertation, an empirical evaluation was performed alongside unit and integration tests. More than 3000 messages/second can be processed and forwarded by the framework, making it suitable for the stated problem. Furthermore, the system proved to be reliable, loading properly third-party developed drivers and recovering from cluster disconnection.

The research and work performed in this dissertation allowed to have a proof-of-concept framework to orchestrate and integrate devices in the AAL4ALL project. Due to its generic architecture, the framework developed may be suitable for other application domains.

Resumo

O paradigma da *Internet of Things* (IOT) em conjunto com a utilização de Redes de Sensores Sem Fios (WSNs) está a revolucionar os chamados *ambientes inteligentes*, tornando real todo um novo mundo de produtos e serviços e melhorando as actividades diárias das pessoas. Desde domótica à vigilância ou Ambient-Assisted Living (AAL), sistemas baseados em WSNs estão a proliferar pelo mundo fora, trazendo novos desafios à sua arquitectura.

Dado o envelhecimento da população mundial, os sistemas de AAL podem providenciar monitorização contínua do estado de saúde de indivíduos, reduzindo a necessidade constante de se deslocarem a instituições de saúde para observação. O AAL4ALL é um projecto financiado pelo governo, envolvendo mais de 30 institutos e empresas, que tem como objectivo construir um ecossistema aberto para AAL. A necessidade de existir uma *framework* que permita que haja *hardware* e *software* de terceiros sobre uma WSN nasceu do facto de qualquer parceiro poder desenvolver os seus próprios produtos e serviços, uma vez certificados.

No cenário previamente descrito, vários sensores podem ser instalados ao longo de casa ou instituições de saúde, para monitorizar um ou mais utilizadores. Alimentados por tecnologias de comunicação de curto alcance como Bluetooth, estes dispositivos requerem redes de sensores para se ligarem perfeitamente, onde quer que o utilizador esteja, mantendo o paciente sempre no raio de um dos nós na rede de sensores, assegurando monitorização ubíqua.

Esta dissertação propõe uma arquitectura para uma rede de sensores genérica, escalável, tolerante à falha e auto-configurável, suportando Bluetooth, permitindo o carregamento dinâmico de *drivers* desenvolvidas por terceiros, a fim de permitir uma evolução contínua do sistema, sem a necessidade de configuração manual.

A solução proposta é baseada no padrão *Publisher/Subscriber*, onde os sensores estão a publicar informação continuamente para uma fila de mensagens na WSN. Os subscritores tanto podem ser actuadores como serviços locais ou na *cloud*, que consumam a informação recolhida. De forma a alcançar concorrência e tolerância à falha, foi adoptado o padrão *Actor model*. Tendo em conta que a utilização de dispositivos de baixa computação, o facto de existirem vários nós distribuídos num edifício foi explorado para construir um *cluster* ponto-a-ponto e tolerante à falha.

Para aferir a validade desta dissertação, foi feita uma avaliação empírica em conjunto com testes unitáros e de integração. Mais de 3000 mensagens/segundo podem ser processadas e reencaminhadas pela *framework*, fazendo-a adequada para o problema proposto. Além disso, o sistema provou ser fiável, carregando correctamente *drivers* desenvolvidas por terceiros e recuperando de desconexão do *cluster*.

A pesquisa e trabalho desenvolvidos nesta dissertação permitiram que haja uma prova de conceito de uma *framework* para orquestrar e integrar dispositivos no projecto AAL4ALL. Dada a sua arquitectura genérica, a *framework* desenvolvida poderá ser adequada para outros domínios de aplicação.

Acknowledgements

To Professor Hugo Ferreira and Tiago Boldt, for their patience, guidance and always helpful advices during this dissertation.

To André Pereira and Luís Carvalho from Fraunhofer AICOS Portugal, for their help tweaking the available sensors and testing all the system.

To my teachers at Ancorensis, in special Indaleto and Chavarria, for all the tremendous knowledge shared in the Informatics course and for encouraging me to pursue ambitious goals.

To Professor Rui Maranhão, one of the best professors I've ever met, for the distributed systems classes which helped me to choose this path for my future.

To my future boss, Elmar Weber, for supporting me in this final rush and giving me all the time to finish this report.

To all my friends who shared the last 5 and half years with me, even some sleepless nights with a lot of coffee.

To Márcia, for all the patience, help and support, for cheering me up when I said that something was impossible to accomplish and for making me feel like I am the best guy in this world.

To all my family, in special to my godparents Toninho and Lúcia and my cousin Bruna, for the countless moments we have shared in the past 22 years.

To Tozé, more than a cousin, a brother and destined to be a great architect. He is one of the most inspiring examples that I try to follow in my life.

To my departed grandparents, António, João and Eva, who certainly would be proud to see their graduated grandson.

To my heroes Florinda (mom), João (dad) and Joana (sister), for all the support, for making me believe in myself and for all the values and lessons taught. Finally, I'd like to thank my grandmother Florinda for raising and educating me and for all the affection over these years.

João Quarteu Alves

"The only way to make software secure, reliable, and fast is to make it small."

Andrew S. Tanenbaum

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Motivation and Goals . . . 2
1.3 Outline . . . 3

2 Problem Statement **5**
2.1 AAL4ALL . . . 5
2.2 Sensor Networks . . . 6
2.2.1 Scalability . . . 6
2.2.2 Auto Configuration . . . 6
2.2.3 Deployment . . . 6

3 State of the Art **7**
3.1 Background . . . 7
3.1.1 *Internet of Things* . . . 7
3.1.2 Wireless Sensor Networks . . . 8
3.1.3 Wireless Technologies . . . 10
3.1.4 Distributed communication . . . 11
3.1.5 The Scala language . . . 13
3.1.6 The Actor Model . . . 13
3.2 Related work . . . 14
3.2.1 TeleCARE . . . 14
3.2.2 ALARM-NET . . . 16
3.2.3 openAAL . . . 17
3.2.4 CAALYX . . . 17
3.2.5 Summary . . . 18

4 Design and Implementation **19**
4.1 Methodology . . . 19
4.1.1 Test-Driven Development . . . 19
4.1.2 Re-using components . . . 20
4.1.3 Deployment . . . 21
4.2 Concepts . . . 22
4.3 High-level Architecture . . . 23
4.4 Implementation Details . . . 24
4.4.1 Concurrency and fault-tolerance . . . 24
4.4.2 Actor hierarchy . . . 25
4.4.3 Interaction with the *cloud* . . . 26

4.4.4 Obtaining configurations . . . 29
4.4.5 Device drivers . . . 31
4.4.6 Clustering nodes . . . 32
4.5 Adopting in AAL4ALL . . . 35
4.5.1 Low-cost computing . . . 35
4.5.2 Communicating with sensors and actuators . . . 36
4.5.3 System deployment . . . 36
4.6 Summary . . . 37

5 Tests and Results **39**
5.1 Unit tests . . . 39
5.2 Integration tests . . . 39
5.3 Real application tests . . . 40
5.3.1 Stress tests . . . 40
5.3.2 Reliability tests . . . 42
5.4 Summary . . . 43

6 Conclusions **45**
6.1 Overview . . . 45
6.2 Summary of the contributions . . . 45
6.3 Future work . . . 46

References **47**

A Twitter Driver **51**

List of Figures

3.1 Typical Wireless Sensor Network architecture. 8
3.2 Multicast example 11
3.3 Broadcast example 12
3.4 Message queue example 12
3.5 *Publisher/Subscriber* example 13
3.6 TeleCARE global approach. 14
3.7 TeleCARE platform architecture. 15
3.8 ALARM-NET platform architecture. 16

4.1 Test-Driven development approach followed. 20
4.2 Sonatype Nexus repository. 22
4.3 High-level architecture of the Wireless Sensor Network. 23
4.4 Wireless Sensor Network node architecture. 24
4.5 Actor hierarchy of the application. 27
4.6 Driver Interface model. 31
4.7 Raspberry Pi model B. 35
4.8 System deployment at home with two nodes in AAL4ALL. 37

5.1 Message processing and forwarding latency from the mock sensor. 41
5.2 Message processing and forwarding latency from the Fraunhofer sensor. 42
5.3 Difference in latency between a mock and the Fraunhofer sensors. 42

LIST OF FIGURES

List of Tables

3.1 Related work summary. 18

5.1 Latency values across the three parts of processing and forwarding data. 41

LIST OF TABLES

Abbreviations

AAL	Ambient-Assisted Living
AAL4ALL	Ambient-Assisted Living For All
CI	Continuous Integration
DNS	Domain Name Server
EU	European Union
GDP	Gross Domestic Product
ICT	Information and Communication Technologies
IoT	Internet of Things
IP	Internet Protocol
LAN	Local Area Network
MAS	Multi-Agent Systems
PC	Personal Computer
PDA	Personal Digital Assistant
SBT	Simple Build Tool
SSP	Secure Simple Pairing
TDD	Test-Driven Development
WSN	Wireless Sensor Network

Chapter 1

Introduction

Smart environments on the top of Wireless Sensor Networks brought a revolution into computer networks. Besides traditional Personal Computers (PCs) there are a set of *smart objects* that could be used to enhance people's daily activities on several domains, such as domotics or ambient assisted living. Thus self-managing and control systems are currently a research trending topic in last years.

These facts can be explored by product manufacturers and service providers creating open ecosystems in order to offer different kinds of services to *end-users*.

1.1 Context

In recent years, the percentage of elderly population has been growing worldwide due to the increased life expectancy. It is expected that by 2060 the number of people with at least 65 years in the European Union (EU) exceed in 100% the number of children [DMSS12]. The costs of health-care are rising as the population is getting older. In 2011, according to the World Bank[1], the US spent 17.9% of their Gross Domestic Product (GDP) on health-care, which means that in nine years the costs arose 4.9%, from the 13% back in 2002 [CCEF09].

The increasing adoption of Information and Communication Technologies (ICT) in our lives decreased the costs of many goods and services, augmenting the productivity [KJ03]. Evolution of technology permitted their general adoption in health-care, changing the way how health-care services are provided, improving life quality and augmenting life expectancy [Var07]. However, contrary to several industries, the cost of technology in health-care is still high [KJ03]. This happens because products created are too expensive, specially to *end-users*, and commonly developed in academic environments [LBC$^+$12], lacking a business model in order to make them marketable.

[1] More details at: `http://data.worldbank.org/indicator/SH.XPD.TOTL.ZS`

Furthermore, there is a rising need to move from the current hospital-centered model which focus on disease treatment to a more affordable one, based on monitoring, prevention and well-being [OMSJ05]. The demand for telehomecare and monitoring systems is growing, in order to avoid long-term hospitalization or nursing home care [CCEF09, BLHG02]. Therefore, the research is looking to provide solutions to monitor seniors' health who lack the financial resources to acquire existing products, combining health-care services, informal caregivers, family and friends with the technological evolution in wireless networking and sensors [OMSJ05].

1.2 Motivation and Goals

Continuous health monitoring systems provided by Wireless Sensor Networks can help to complete this transition to a model focused on the patient [WVD+06]. Capturing real-time data such as temperature, blood pressure or insulin levels could improve both doctors ability to diagnose the patient's condition and the assistance provided by caregivers or family members who can be alerted in case of a possible health problem. There is also a real advantage to the patient himself, since he can better control his own health state. Considering [Org11], heart diseases are the largest cause of death worldwide, representing 30% of all global deaths and can be prevented if detected early. Wellness issues associated with financial constraints and the need to free up beds in the hospitals are increasing the demand for continuous monitoring systems using wireless based sensors.

Monitoring and automated control of devices on top of distributed networks has been a trending topic in the past few years. While several studies have been conducted in this area, the lack of a business model and the high costs of hardware lead to nonmarketable products, mainly when aiming at home users. The existing systems address the problem based on its final use, and there is a lack of a more general solution which could solve several problems of different domains.

Motivated by this, we wanted to develop a framework for managing a cluster of computers (nodes) in a Local Area Network (LAN), extensible with behavior for either control or monitoring third-party devices. These devices can be connected over wireless protocols such as Wi-Fi, ZigBee or Bluetooth. However, all of these technologies share a common problem: they are short-ranged, which is the reason to build a cluster of nodes. We need to ensure coverage across a building without data loss, so it is essential to have more than one node, allowing the user to walk freely around the house without the danger of losing any monitoring data.

It is expected that nodes discover each other and cluster automatically, distributing the computation over the available nodes in a cluster without a single point of failure. The cluster should also provide the ability to be extended with processing capabilities over the acquired data and propagate it to *cloud* services. Aiming at providing low-cost clusters, initial nodes will run on Raspberry Pi[2]'s. Furthermore, the resulting application should be generic enough to be exported to other application domains.

The main goals of this dissertation are listed bellow:

[2] More details at: `www.raspberrypi.org`

1. **Automatic configuration of each node.** When a new node is plugged into the electrical power, it is expected that it automatically installs all the required software and drivers from an *online* repository. After that, the main application should start, providing interfaces to communicate with sensors and actuators around the house.

2. **Acquire and process data from different sources, controlling the connected devices.** This type of systems have several different data sources (e.g., blood pressure and cardiac rhythm sensors, light sensor, etc) and the application should manage the devices connected to each node and search continuously for new devices, in order to minimize the downtimes. Different sources also means different manufacturers and different drivers, which arose the need of loading hardware drivers dynamically, in *run-time*, without affecting the rest of the system.

3. **Detect peer nodes on the network and cluster with them automatically.** The application should automatically detect peer nodes and be able to distribute tasks across them, balancing the workload of the system.

1.3 Outline

This report is structured into three different parts, with the following structure:

Chapter 2: "Problem Statement" explains in more detail the problem to be addressed in this report.

Chapter 3: "State of the Art" gives an overview about the Internet of Things, Wireless Sensor Networks and about the wireless technologies which support them. Then we will describe related projects in the AAL area, analyzing and comparing them with the desired functionality of this dissertation's project.

Chapter 4: "Design and Implementation" describes our approach to the problem. It is also described the methodology used as well as the solution's high-level architecture and some technological decisions.

Chapter 5: "Tests and Results" presents our system evaluation according to the tests described in this chapter.

Chapter 6: "Conclusions" will present the conclusions of our work, reviewing our goals and achievements and presenting future work to be done in the context of this dissertation.

Chapter 2

Problem Statement

Aging and associated health degradation tend to reduce autonomy in older adults, who require frequent monitoring [OMSJ05, WVD+06, Var07, LBC+12]. New technologies can be used to continuously monitor their health conditions in an attempt to early detect dangerous conditions in their health status.

By acquiring data related to the user's vital signs, detecting falls or through location sensors, caregivers can remotely assure the patients' status. Thus, it is possible to reduce the need for frequent visits to the doctor, enhancing citizens' life quality. Such technology can also simplify the diagnose and follow up on patients that require constant monitoring due to their health conditions.

2.1 AAL4ALL

AAL4ALL (Ambient-Assisted Living For All) [AAL] is a nation-funded project, developed by a consortium of over 30 partners, including universities, research institutions and industry, that aims to create an ecosystem of products and services for Ambient-Assisted Living (AAL) in Portugal. By providing an open ecosystem, any partner can join in and provide his products, once certified. The products can be either a sensor, an actuator or a virtual service. People can have their own customized experience by purchasing or subscribing only the required equipment and services that better fits them. A large scale trial will validate the project at the end, with tens of users adopting the initial services and equipment for a prolonged amount of time.

Resulting from the open nature of the ecosystem, several sensors and actuators would be available at the patient's house through Bluetooth. Using this wireless protocol, there is the need to provide a sensor network which must be capable of manage the communication with it as described in the following section.

The global architecture of this project can be divided in two parts: a lower layer responsible for capturing data from the patient and a top layer which receives this information and forwards it to the appropriate service. This top layer keeps a list of service subscriptions per user and allow them

to subscribe to data generated by the user. For this to happen, the bottom layer requires an WSN to capture and forward data to the cloud. The developed solution is presented in chapter 4.

2.2 Sensor Networks

The need of a WSN arose from the AAL4ALL goal to provide an ecosystem of services and equipment for monitoring patients, either in their homes or health-care institutions. Focusing on the indoor scenario, the patient is expected to walk freely in the several divisions and Bluetooth, which sensors rely on, is a short ranged technology [LSS07]. In order to continuously gather data from each patient, a WSN should be deployed to avoid any lack of coverage all around the house for both communication protocols, which can be expensive. Considering this project, it is vital to have a WSN node that communicates either with ZigBee and Bluetooth devices. Furthermore, there are other concerns, such as the scalability, auto configuration of the system and software deployment which are explained in subsections 2.2.1, 2.2.2 and 2.2.3, respectively.

2.2.1 Scalability

The system must ensure full coverage over a building and the number of sensors and actuators deployed may be dozens or hundreds, depending on the building. Due to the use of short ranged technologies [LSS07], it is expected the coexistence of more than one AAL4ALL node, in order to achieve a lossless and fault-tolerant system. Furthermore, this may permit a distribution of tasks such as data processing and drivers management over the available WSN nodes.

2.2.2 Auto Configuration

Most of the people do not have a deep technological knowledge to install and configure this kind of system. The AAL4ALL project is intended to be used by everyone. The installation should be simple as plug each node to the electrical power. After that, it is expected that each node auto configures itself, connecting to the network and downloading the corresponding configurations, such as hardware drivers and software, from a repository located on the *cloud*.

2.2.3 Deployment

The deployment of the software application and all its dependencies should be transparent to the user. This factor is important because it should allow to update the existing software – both third-party software in which our application rely on and the application itself – without any external intervention. Thus, it is expected that it will only be necessary to configure the operative system once and it will automatically update itself with the proper software.

Chapter 3

State of the Art

This chapter analyzes the state of the art of Wireless Sensor Networks and how are they used in monitoring and control problems.

In Section 3.1 is presented the background of the dissertation. Section 3.1.1 introduces the *Internet of Things* and how it is related to smart and self-managed environments. In Section 3.1.2 it is briefly described the global architecture of a WSN. Section 3.1.3 enumerates wireless technologies related to WSNs, showing their main characteristics. Section 3.1.4 discusses distributed communication patterns and its advantages or disadvantages in WSN. In Section 3.1.5 Scala language is introduced. Section 3.1.6 introduces the Actor Model and its main advantages building a fault-tolerant system.

Section 3.2 shows several monitoring and self-managing systems projects with a special focus in AAL. Finally, in Section 3.2.5, the studied projects are summarized and compared with the goals of this dissertation.

3.1 Background

This section presents a background on the *Internet of Things* (Section 3.1.1) and Wireless Sensor Networks (Section 3.1.2), focusing key concepts addressed by this dissertation. Moreover, a discussion about related projects is presented in Section 3.2 as well as a summary of the literature review and some conclusions in Section 3.2.5.

3.1.1 *Internet of Things*

According to [AIM10], the *Internet of Things* is a novel paradigm which concept is to have a network of interconnected *smart objects* forming pervasive computing environments [MSPC12], through unique addressing schemes. Any object, such as mobile phones, sensors or actuators, should be able to join the network and cooperate with its neighbors to achieve common goals through a distribution of different tasks [CRMS09, AIM10]. By embedding computational capabilities in objects, the IOT will bring new opportunities to the ICT sectors that need smart environments [HSW+00], such as health-care or domotics [ANLR10].

3.1.2 Wireless Sensor Networks

The use of WSNs is commonly associated to monitoring and control problems in several domains such as health [WVD+06], military [HKL+06] and environmental [WAJR+05] systems.

Wireless Sensor Networks are extremely important in the IOT paradigm, since they can encapsulate a group of heterogeneous objects into a transparent system [ANLR10]. A common global architecture of a WSN is shown in figure 3.1. Each sensor has the capability to collect and route data to the network, which can process the data or/and send it to the *cloud*. Moreover, the connection between a sensor and a WSN is not static, meaning that a sensor is able to switch from a WSN to another, communicating with several WSNs over the time.

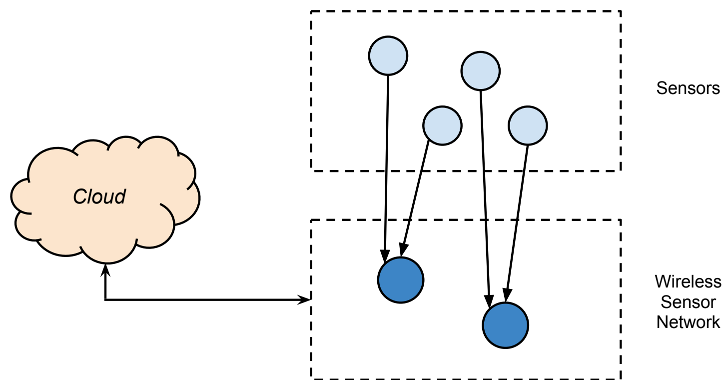

Figure 3.1: Typical Wireless Sensor Network architecture.

3.1.2.1 Applications

Although WSNs were initially designed to serve military purposes, nowadays it is possible to think about a huge number of applications. Assisted driving, mobile ticketing, industrial plants monitoring, object tracking, environmental monitoring or habitat study are possible applications of WSNs [AIM10, SGAV12]. Also, futuristic applications can be thought, such as robot taxis [AIM10] that are aware of traffic movements and have several proximity sensors to avoid collisions either with cars or other objects on the road.

3.1.2.2 Challenges

Taking into account the stated before applications, building WSNs rises some challenges:

Resource constraints

Sensors and actuators are limited and have restricted computational capabilities and battery supplies [GH09]. This fact leads to other issues and challenges, such as the implementation of efficient data transfer protocols or security mechanisms in these devices [AIM10, ABRV12].

Quality-of-Service

Quality-of-Service in WSNs can be categorized in two perspectives *application-specific*and *network-specific* [YIE11]. The first one focuses in the application requirements such a lifetime, coverage, deployment [YIE11] and sensing accuracy between the data reported to WSN nodes and what is really occurring [GH09]. *Network-specific* perspective takes into consideration network characteristics such as latency, packet loss and reliability [YIE11]. Sensor data is typically time-sensitive and wireless connections may change over time due to the use of short-ranged technologies or interferences [GH09].

Scalability and self configuration

WSNs may scale to thousands of nodes, requiring more flexible and scalable solutions [SGAV12], in order to support heterogeneous application domains over the same architecture [GH09]. Since thousands of sensors can be randomly deployed across a building or even in a physically unreachable area, they are required to operate by themselves. These networks should have the ability to recover from failures, reconfiguring the system or restarting modules without affecting the applications that are running upon [GH09].

Security

WSNs are specially vulnerable networks, because communications rely on wireless technologies, which can be problematic due to eavesdropping, jamming, denial of service attacks or injecting malicious traffic [AIM10, ABRV12]. As stated before, thousands of sensors may be deployed across buildings, which make them vulnerable to be captured or tampered [GH09, AIM10]. Moreover, since the adopted devices are commonly low-powered and with limited computational capabilities it is difficult to implement complex and efficient security layers [AIM10, ABRV12].

Privacy

The increasing adoption of WSNs to daily activities leads to a major concern on privacy. Sensors are continuously transmitting sensitive data about people such as their daily activities or health status [GH09, AIM10, ABRV12]. So there is a need to prevent malicious applications to get data that they are not authorized to access, through privacy policies. These must ensure that it is not possible to link people's data to their identities and the deletion of the collected data as it is not need anymore for controlling or monitoring purposes [AIM10].

It is very challenging to build an architecture that meets all stated before goals simultaneously. As stated before, resource constraints can be a barrier to the Quality-of-Service or to implement security mechanisms. Alongside with self configuration, these aspects are the most relevant to the objectives pursued in this dissertation.

3.1.3 Wireless Technologies

Since there is an increasing use of wireless technologies on the top of sensor networks, it is necessary to study what are the technologies used and why are they used in several WSNs. Bluetooth (Section 3.1.3.1), ZigBee (Section 3.1.3.2) and Wi-Fi (Section 3.1.3.3) as well as other wireless technologies (Section 3.1.3.4) are presented below.

3.1.3.1 Bluetooth

Bluetooth is a wireless technology for exchanging data over short distances, originally created in 1994 at Ericssson. The devices are connected through a *piconet*: an *ad-hoc* network which allows to have a *server* device to interconnect with up to seven *client* devices [LSS07]. However, this system allows up to more 255 inactive clients, through a *scatternet*, which is basically a cluster of two or more *piconets*. In a *scatternet*, the server can turn active one device at any time, turning one of the current seven clients inactive [LSS07].

To share information between two devices they must be paired. Prior to Bluetooth v2.1, there is a legacy pairing method, where each device must enter a PIN code and the pairing is only successful if both devices enter the same code. After Bluetooth v2.1, it was implemented the Secure Simple Pairing (SSP) which uses a form of public key cryptography in order to avoid man-in-the-middle attacks and simplify the pairing process.

3.1.3.2 ZigBee

ZigBee is a specification built upon the IEEE 802.15.4 standard for a suite of high level communication protocols used to create Wireless Personal Area Networks (WPAN) for supporting simple and low-cost devices that consume minimal power [BPC+07, LSS07].

Though low-powered, ZigBee devices are often used in mesh network, allowing more than 65000 cell nodes [LSS07] and making possible to transmit data over longer distances as long as each node can communicate with other.

3.1.3.3 Wi-Fi

Wireless Fidelity, Wi-Fi, is a popular technology built upon the IEEE 802.11 standards for Wireless Local Area Networks (WLAN), that allows to exchange data without wires [LSS07]. Any Wi-Fi enabled device can connect to the Internet via an access point, which could be problematic due to its location, there could be many dead zones without coverage in a home [AWW05]. Despite the wider communication range of Wi-Fi protocol, the cost/power consumption rate is not as good as in Bluetooth or ZigBee [PB10].

3.1.3.4 Other technologies

There are other wireless technologies, such as UWB or IrDA which could be used by AAL4ALL partners on their sensors and actuators.

UWB, short for Ultra-Wideband, is a radio technology built upon the IEEE 802.15.3 standard, which may be used at a very low energy level for short-range high speed wireless connections [LSS07]. Although this technology is not appropriate to a WSN due to its high complexity and the unsuitable wide bandwidth modulation [HF08, PB10].

Infrared Data Association (IrDA) is a group founded in 1993 which defines a set of protocols for wireless infrared communications that uses point and shoot principles to transfer data between low-cost, low-powered and short-range devices [WML95, Wil00]. IrDA does not require to pair devices in order to communicate, but it needs a direct line of sight.

3.1.4 Distributed communication

Supporting communication between devices through a WSN has some particularities. Many entities are interested in receive data from several sources. Therefore, distributed communication patterns, such as Multicast (Section 3.1.4.1), Broadcast (Section 3.1.4.2) and Message queues(Section 3.1.4.3) are discussed in this section.

3.1.4.1 Multicast

As seen in figure 3.2 multicast is the delivery of a message or information to various, but usually not all, hosts over a computer network [Tan02]. Multicast is most commonly implement over the Internet Protocol (IP) and it is often used when distributing real time audio and video to the set of hosts which have joined a distributed conference. Although there is no connection setup or tear-down and messages are only received by hosts that are interested in them, multicast is not reliable since messages may be lost [Tan02].

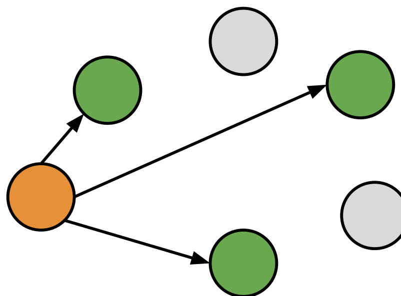

Figure 3.2: Multicast example. A message is sent over the network but only the interested hosts will receive it.

3.1.4.2 Broadcast

Broadcast is the term used to describe communication where a message or information is sent from one host to all other hosts within the network, as shown in figure 3.3 [Tan02]. This could be a possible solution to send information across all the devices in WSNs but there are come constraints

such as the heaviness of the broadcast traffic generated when deploying thousands of devices. Moreover, data consumers have different information interests, so it could be a huge waste of bandwidth and energy.

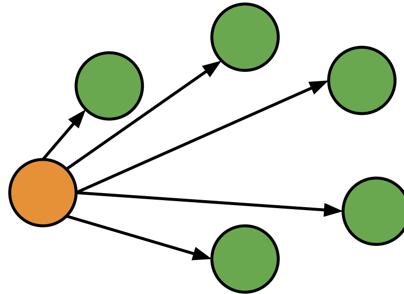

Figure 3.3: Broadcast example. A message is sent over the network and all the hosts will receive it.

3.1.4.3 Message Queues

Message Queues are software components that allow asynchronous communication between two software processes or threads. This fact arises the need of having a third entity: a message broker which is an intermediary that receives messages from one or more senders and route them to one or more receivers. Figure 3.4 shows an example of a message queue. There is a producer P and a consumer C and the message flow goes through a message queue named “hello”.

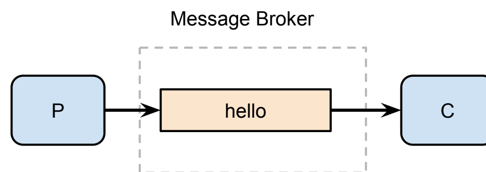

Figure 3.4: Simple message queue example. Adapted from `http://www.rabbitmq.com/img/tutorials/python-one.png`

When working with message queues there are several configuration options, such as: i) durability, ii) delivery and routing policies, iii) acknowledgment of a received message or iv) time-to-live of the messages.

Publisher/Subscriber Publisher/Subscriber is a data-centric communication approach based on message queues, where senders of messages, called *publishers*, do not need to know the receivers addresses, called *subscribers* [BMR+08]. Instead, published messages are characterized into classes, without knowledge of what subscribers there may be. Every publisher can chose in what topic he will send the message and the subscribers can chose in what topic they are interested

in [BMR$^+$08]. As shown in figure 3.5 the message broker routes the information from publishers to subscribers as well as it manages the list of publishers and subscribers. In the addressed problem, there could be topics for each different components grouped by classes (e.g., domotics, chardiac sensors, etc). This pattern provides greater network scalability, since it allows decoupling between publishers and subscribers.

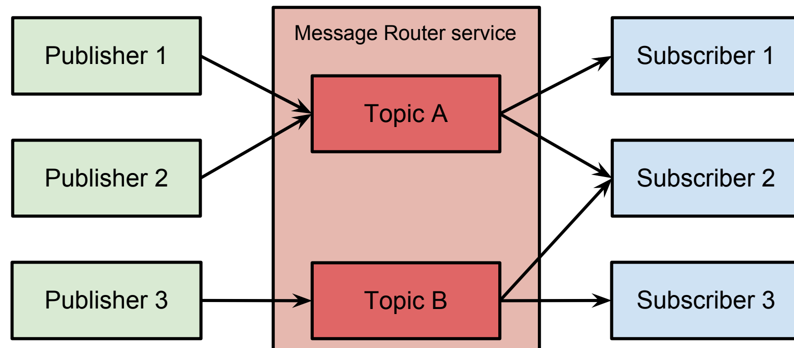

Figure 3.5: *Publisher/Subscriber* pattern and message routing example.

As stated in Section 3.1.2 it is possible that sensors switch from a WSN to another, which means that devices may change their address over the time. The *Publisher/Subscriber* pattern is suitable for this dynamic topology, due to the fact that publishers and subscribers do not need to know each other addresses.

3.1.5 The Scala language

Scala is an object-functional programming language created by Martin Odersky and it is compiled to Java bytecode and runs over the Java Virtual Machine, which allows to using all of the Java libraries directly in Scala code [OSV08]. This fact make Scala a good candidate language to address this dissertation's problems due to its compatibility and its portability, since the Java Virtual Machine is available for the most existing architectures and operative systems.

3.1.6 The Actor Model

The Actor Model was first introduced by Hewitt et al. [HBS73] and then it was improved by Agha [Agh86] and it was a new approach to the concurrency problems caused by threading and locking. In the actor model each object is an actor which is comprised by three parts:

An address where the messages are sent to.

A mailbox where the received messages are buffered.

A behavior which will be applied to the messages in the mailbox, one at time. There are three operations that an actor needs to support: i) **send** messages to other actors; ii) **create** new actors; iii) **assume new behavior** for the next message to be received.

All the communication in this model happens by exchanging asynchronous messages between actors. Since actors never share state they do not need to compete for locks in order to access to shared data. Moreover, it is easier to isolate components, which is an advantage while dealing with fault-tolerant systems.

3.2 Related work

This section describes different studies and projects about self-managed monitoring and control systems. It is divided in four subsections where four different projects are presented and analyzed, showing limitations and differences when compared to the dissertation's goals.

3.2.1 TeleCARE

TeleCARE is a project which aims to build a configurable and generic framework focused on virtual communities for elderly support [CMA04, CMRO04] (Figure 3.6). These virtual communities connect elders who are at home with their relatives and care institutions through a three-level architecture, separated in a basic platform and specialized components [CMA04] (Figure 3.7).

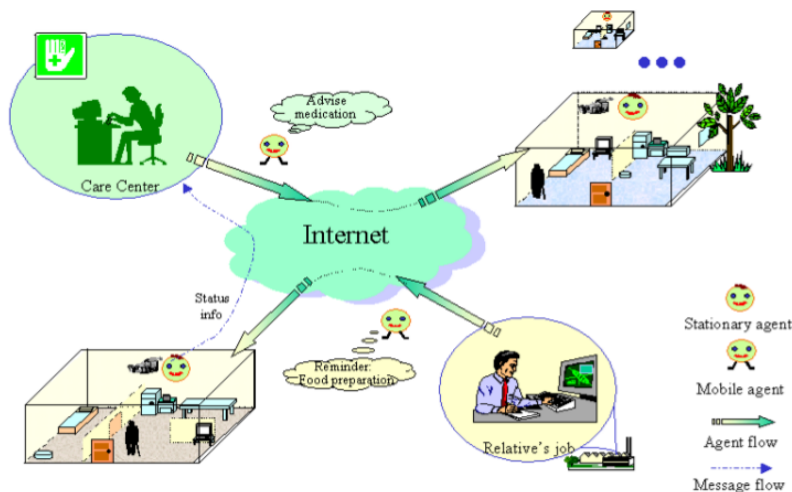

Figure 3.6: TeleCARE global approach.

The basic platform is constituted by an external layer for device abstraction and a Multi-Agent System (MAS) layer which is the core of the TeleCARE systems [CMRO04]. The MAS layer is responsible for two critical features of the framework: the inter-communication between agents, through *FIPA ACL*[1] messages and the agent management and failure recovery [CMA04].

[1] *FIPA ACL* message structure specification: `http://www.fipa.org/specs/fipa00061/SC00061G.pdf`

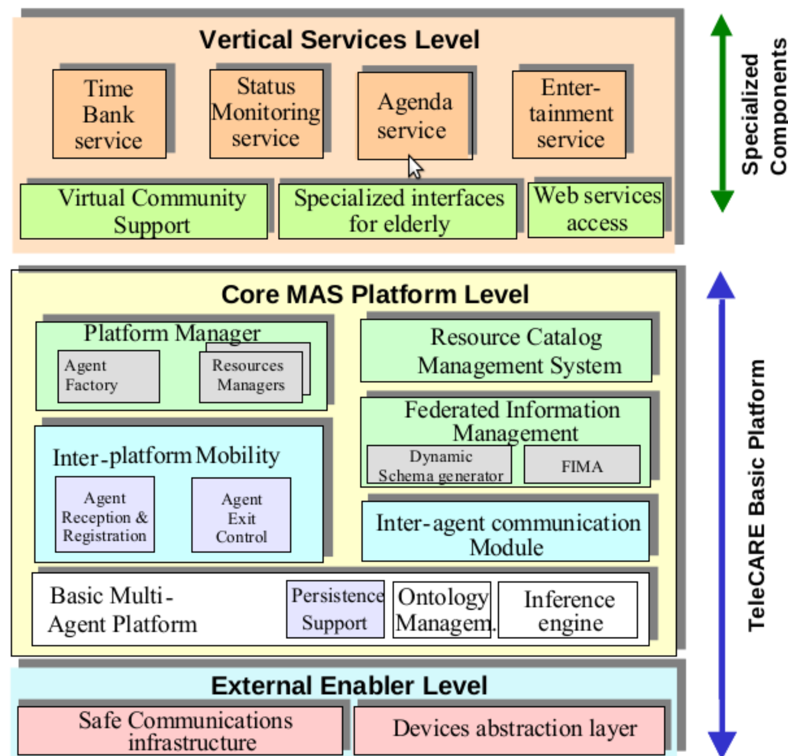

Figure 3.7: TeleCARE platform architecture.

On the top of the basic platform, it is possible to have several specialized components such as interfaces for monitoring systems or web services to collect and analyze data [CMA04].

3.2.1.1 Discussion

TeleCARE authors present a generic architecture for AAL, providing abstraction layers both for hardware and software, through a MAS core. However, it is not specified how the framework deals with third-party hardware drivers.

The authors do not mention if there is any automated testing protocol before their deployment or if there is an online repository where the drivers can be downloaded or updated. Also, in a system like this it is possible that several versions of an hardware device could coexist, but the problem is not addressed by the authors.

Despite the distributed and fault-tolerant environment presented, TeleCARE does not cluster the available nodes, which could allow load balancing and improve the power consumption of the system. Furthermore, there is no reference to the hardware requirements of the framework or if it is prepared to low-cost and low-powered devices. This is a relevant issue when a solution aims to reach final consumers, like elderly citizens and their relatives.

3.2.2 ALARM-NET

ALARM-NET is a WSN for AAL and residential monitoring, developed at the University of Virginia [WVD+06]. The presented architecture (Figure 3.8) relies on a Body Area Network (BAN), which is a wearable piece that aggregates a set of wireless sensor devices, tailored for each patient. Moreover, several sensors, such as temperature, motion and light, are deployed all around the house, creating a multi-hop wireless network [WVD+06].

The gateway between the WSN and IP networks is performed by a group of nodes called AlarmGate nodes, which permit the inter-communication between user interfaces and the back-end. Finally, the user interfaces allows the patient, relatives or caregivers to access the sensors data through a PC or a PDA [WVD+06].

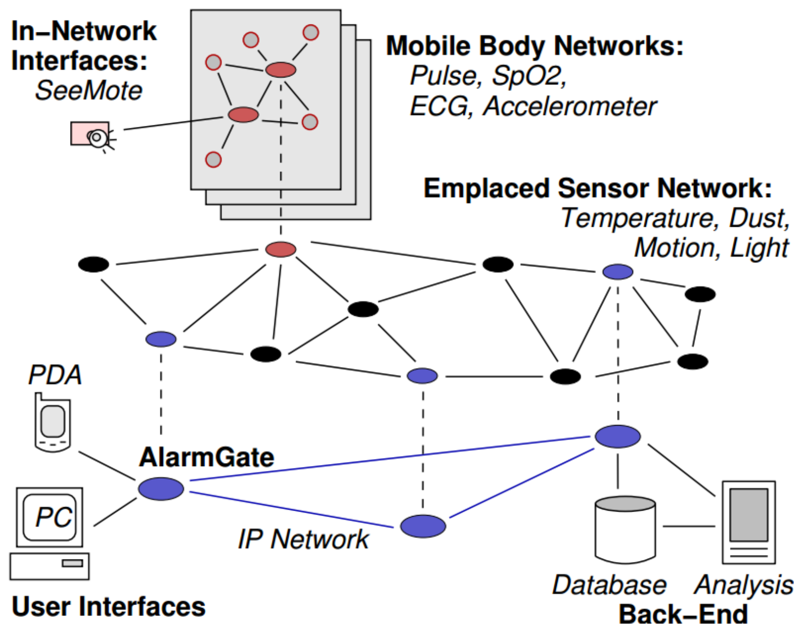

Figure 3.8: ALARM-NET platform architecture.

3.2.2.1 Discussion

Despite the innovative Circadian Activity Rhythm analysis module, which could improve the system power consumption through the learning of citizens' daily activities, ALARM-NET architecture has some problems. Since body sensors are deployed through unique wearable devices and elder's health condition may change over time, is difficult to adapt this system to new situations.

The authors do not specify how the installation and configuration is performed and if it is possible to update the software without a specialized technician. Additionally, ALARM-NET is a closed architecture, without focus on support third-party devices or software, which limits the potential of the solution.

3.2.3 openAAL

OpenAAL is an open-source middleware for AAL solutions, built on top of an OSGi framework [All07], that allows easy integration and communication between services [WSO+10]. The openAAL middleware has three main components: Context Manager, Procedural Manager and Composer.

Context Manager is an ontology-based information storage that captures sensor information and user input [WSO+10]. Procedural Manager decides if the system should or should not inform the user or caregivers about a situation, based on the Context Manager information. Finally, Composer module finds and executes virtual services (e.g.: TV alert, send SMS, etc) upon request. As an example, if an assisted person is leaving his home and the Context Manager notices that the tv is turned on, the Procedural Manager module will notify the Composer module, in order to find a way to notify the person.

3.2.3.1 Discussion

The openAAL framework has abstraction layers both for hardware devices and third-party software. Although, its configuration is complex and it is built on the top of an OSGi framework which does not address the problem of providing fault tolerance support for bundles. Furthermore, the latest available source-code is from 2010 and the project's documentation is not complete.

3.2.4 CAALYX

CAALYX (Complete Ambient Assisted Living Experiment) is a project funded by the European Commission under the AAL Joint Programme[2] [RMFJ+11]. CAALYX is divided in three main systems: Home, Mobile and Caretaker.

The home system is a monitoring system which takes advantage of the existing TV set to interact with the user, helping him to live independently at home [RMFJ+11]. To capture his vital signs, the elder must use the Wearable Light Device (WLD), a unique wearable piece which concentrates several wireless sensors in a BAN. The Mobile system controls the BAN, reducing the dependency from a central system and it has a local reasoning system which detects possible elder's health problems [RMFJ+11].

The intercommunication between assisted people, family and caregivers is performed by the Caretaker system. The Caretaker system is a logic layer which offers services such as monitoring and raises alerts, depending on elder's health condition [RMFJ+11].

3.2.4.1 Discussion

Unlike other discussed projects, CAALYX project has a major concern about how the elder will use the system, trying to make it usable to old-aged people. The authors shows preoccupation to build a platform that can have full functionality in low-cost and low-powered device.

[2] AAL Joint Programme official website: `http://www.aal-europe.eu/`

On the other hand, CAALYX sensors are deployed in a unique hardware piece, with limited sensors which means that it is difficult to adapt the BAN to new health conditions and elder's requirements. Since CAALYX relies the control of the BAN in a mobile phone, there is a problem associated with the battery. Sometimes the assisted person do not remembers to charge the mobile phone, compromising the system. Furthermore the lack of a software abstraction layer leads to a closed environment with limited applications.

3.2.5 Summary

In the last years, WSNs are being used to solve remote monitoring and control problems, more specifically in AAL. However, as it is possible to see in table 3.1, none of the studied projects presents solutions to all the problems raised by this dissertation.

OpenAAL (3.2.3) provides a configurable framework with abstraction layers both for hardware and software, but it lacks an easy way to configure the all system and it has performance problems. On the other hand, CAALYX (3.2.4) shows a more user-friendly solution, easy to configure and using the user's TV set but with limited hardware and software solutions.

Table 3.1: Related work summary.

	TeleCARE	ALARM-NET	openAAL	CAALYX
Hardware and Software abstraction	Yes	No	Yes	No
Easy configuration	No	No	No	Yes
Suitable for low-powered and low-cost devices	No	Yes	No	Yes
Drivers and software update dynamically	No	No	Yes	No
Fault-tolerant	Yes	Yes	No	Yes
Device clustering	No	No	No	No

Chapter 4

Design and Implementation

This chapter presents the solution to the problems addressed by this dissertation. Basic concepts in which the solution was built upon are explained in Section 4.2. The high-level architecture is shown in Section 4.3. Section 4.1 shows the methodology used in the development of this dissertation. In Section 4.5 is explained the framework adoption in the AAL4ALL project. Section 4.6 synthesizes the solution and discusses eventual portabilities to other domains.

4.1 Methodology

The methodology followed in this dissertation is described in this section. Section 4.1.1 describes the implementation methodology. In Section 4.1.2 we describe how the software packages are re-used. The deployment strategy used are described in Section 4.1.3.

4.1.1 Test-Driven Development

During the implementation phase we adopted an agile process, writing tests for each created component and releasing versions periodically. In order to do that, a *Continuous Integration* (CI) environment was setup. As shown in figure 4.1, the development and test cycle occurred as follows:

1. Write the code for a feature
2. Write unit tests for a given feature
3. Commit the changes to the version-control repository
4. Run all the tests on a CI server
5. Refactor and repeat the process

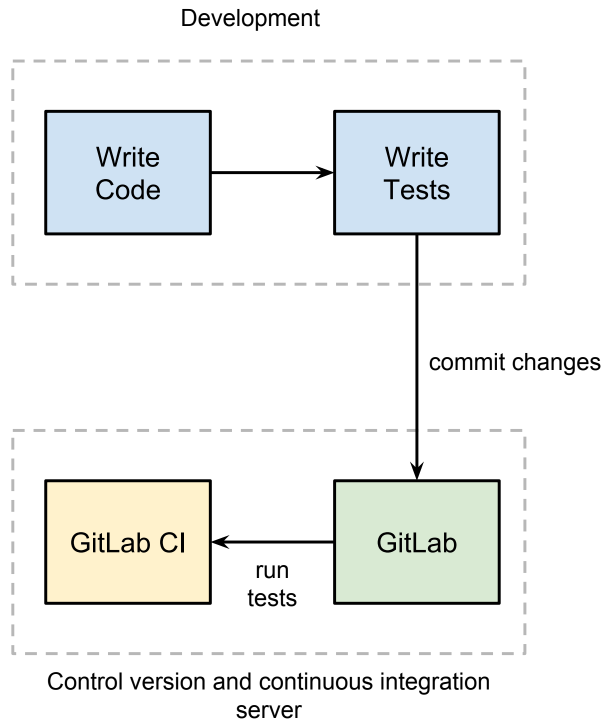

Figure 4.1: Test-Driven development approach followed.

On the server-side, they were installed GitLab, an open-source source-code manager that allows to create teams, roles, projects and keep track of them. GitLab[1] is built upon Git[2], a distributed source control manager. Furthermore, GitLab CI was also installed in order to integrate with the existing projects on GitLab and run the proper tests for each one.

4.1.2 Re-using components

In this dissertation we adopted a strategy to re-use the created software components. In the design phase we identified the main components of the application and decoupled them in order to have a more flexible architecture and to test each component separately. Moreover, this way it is possible to re-use the components produced in other projects.

4.1.2.1 SBT

SBT[3] is the abbreviation of "simple build tool" and it is an open-source build tool for Scala and Java. Using it we can write an SBT file with the library dependencies of a given project and a repository

[1]GitLab's official website: `http://gitlab.org/`
[2]Git's official website: `http://git-scm.com/`
[3]SBT official website: `http://www.scala-sbt.org/`

to publish our own projects. An example of importing software components is shown in Listing 4.1 and another one of publishing software components is shown in Listing 4.2.

With this build tool we can create one project for each software component, publish them in a repository and re-use them after, making an application which aggregates the software components. This way we can iterate and test each component without breaking the final application.

```
1 libraryDependencies ++= Seq(
2   "pt.inescporto" %% "zeroconf" % "0.23"
3 )
```

Listing 4.1: Re-using software components with SBT.

```
1 credentials += Credentials(Path.userHome / ".sbt" / "credentials")
2
3 publishMavenStyle := true
4
5 publishTo := {
6     Some("aal4all"  at "http://cathedralgotix.inescporto.pt:83/nexus/content/
          repositories/aal4all")
7 }
```

Listing 4.2: Publishing software components with SBT.

4.1.2.2 Sonatype Nexus

As stated before in Section, with SBT we can define a repository to publish the software components. Sonatype Nexus[4] provides a central point for management of binary software components and their dependencies, so every developer with access to the repository can re-use the available software components. Figure 4.2 shows an example of a repository with several software components available.

4.1.3 Deployment

Setting up the application and all its dependencies can be a complex task, so we wanted to make this process easier and provide a simple initial setup.

Using a SBT plugin named “SBT Native Packager”[5] is possible to bundle up Scala software for many operative systems such as Debian[6], Red Hat[7] or Microsoft Windows[8]. Listing 4.3 shows an example of how to inject the OpenJDK[9] dependency for Debian operative system in a SBT file.

[4] Sonatype Nexus official website: http://www.sonatype.org/nexus/
[5] SBT Native Packager github’s page: https://github.com/sbt/sbt-native-packager
[6] Debian’s official website: http://www.debian.org/
[7] Red Hat’s official website: http://www.redhat.com/
[8] Microsoft Windows’ official website http://windows.microsoft.com/pt-pt/windows/home
[9] OpenJDK’s official website: http://openjdk.java.net/

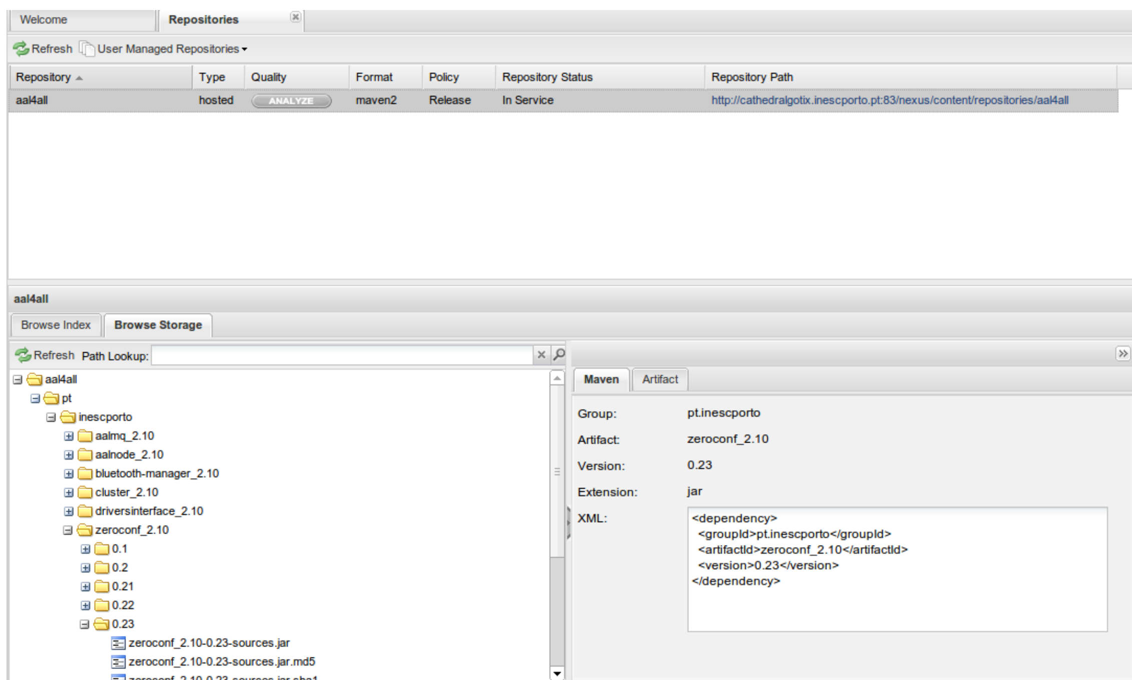

Figure 4.2: Sonatype Nexus repository.

```
1 debianPackageDependencies in Debian ++= Seq("openjdk-7-jre (>= 7)")
```

Listing 4.3: Injecting software dependencies with SBT.

4.2 Concepts

In a Wireless Sensor Network it is possible to identify two fundamental features: to collect data from sensors and to deliver the collected data, either locally processed or sent to an external service, hosted on the *cloud* [HTSC08, BMKK12]. Sensors are publishing information continuously so that actuators and virtual services can subscribe to the gathered data. Hence, for communication between devices we adopted a *Publisher/Subscriber* pattern, a data-centric communication approach where publishers and subscribers do not need to know each other destination or addresses in order to communicate. This particularity solves the problem of connecting and disconnecting sensors, which in a dynamic system like this can be connected to different nodes in different moments, having different addresses too.

As seen in figure 3.5, published messages are categorized by topics, due to the fact that publishers do not know who and how many are the data subscribers. In the same way, subscribers can only subscribe a subset of all the exchanged messages. In order to distribute the information to the subscribers, WSNs must must run a message routing service.

4.3 High-level Architecture

Sensors and actuators are low-powered devices and they cannot interact directly with the upper system, in the *cloud*. The WSN nodes are needed both to orchestrate the data-flow between publishers and subscribers and to manage the connections between the Bluetooth-enabled devices and the upper system. Moreover, Bluetooth is a short-ranged technology, which means that in order to achieve full coverage inside a building and avoid data loss situations, we need to deploy more than one WSN node. Moreover, since the deployed nodes are meant to be low-cost and low-computing devices it makes sense to take advantage of the stated above nodes and build a cluster of nodes, interchanging data and performing load balancing and task distribution across them.

The high-level architecture of the solution is presented in figure 4.3. As stated in Section 4.2, publishers (sensors) and subscribers (actuators and services) are the two main entities. Subscribers can receive information either from local services, running on the top of the WSN node cluster, or information directly from the *cloud*. In this dissertation, the focus is maintained in the WSN nodes.

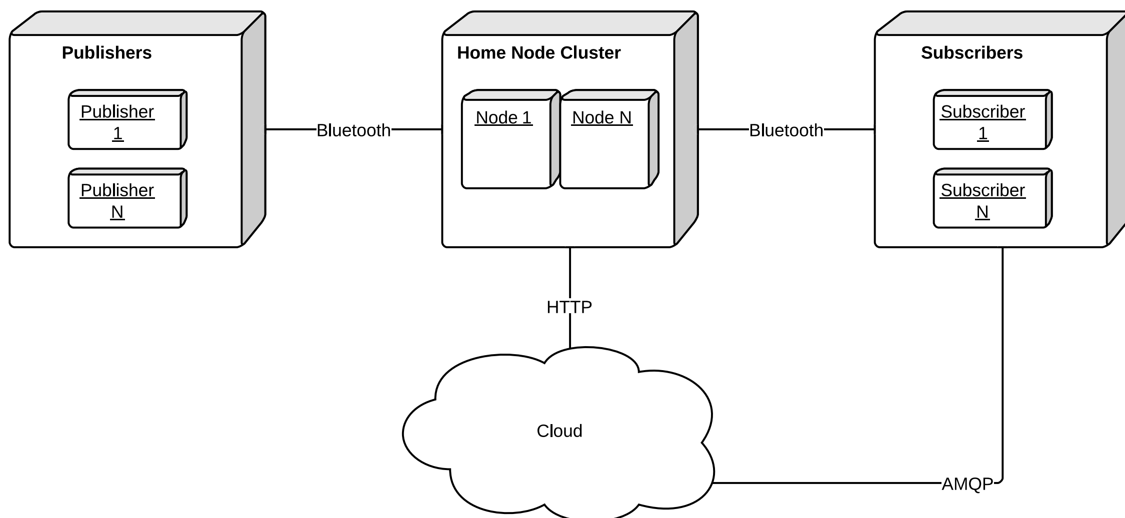

Figure 4.3: High-level architecture of the Wireless Sensor Network.

As seen in figure 4.4, there are three major components in a WSN node. On the top, there is an AMQP[10] client which communicates with the upper levels in the global architecture (figure 4.3), publishing acquired data and subscribing to relevant commands to control local actuators.

The auto configuration (section 4.4.4) service is responsible to properly configure each node, receiving new drivers to dynamically load them. There is also a cluster service, which allows clustering of the WSN nodes, as explained in Section 4.4.6.

Finally, the communication component is responsible both for managing the connected devices as well as discover new devices in its range.

[10] Advanced Message Queue Protocol. More details at: `http://www.amqp.org`

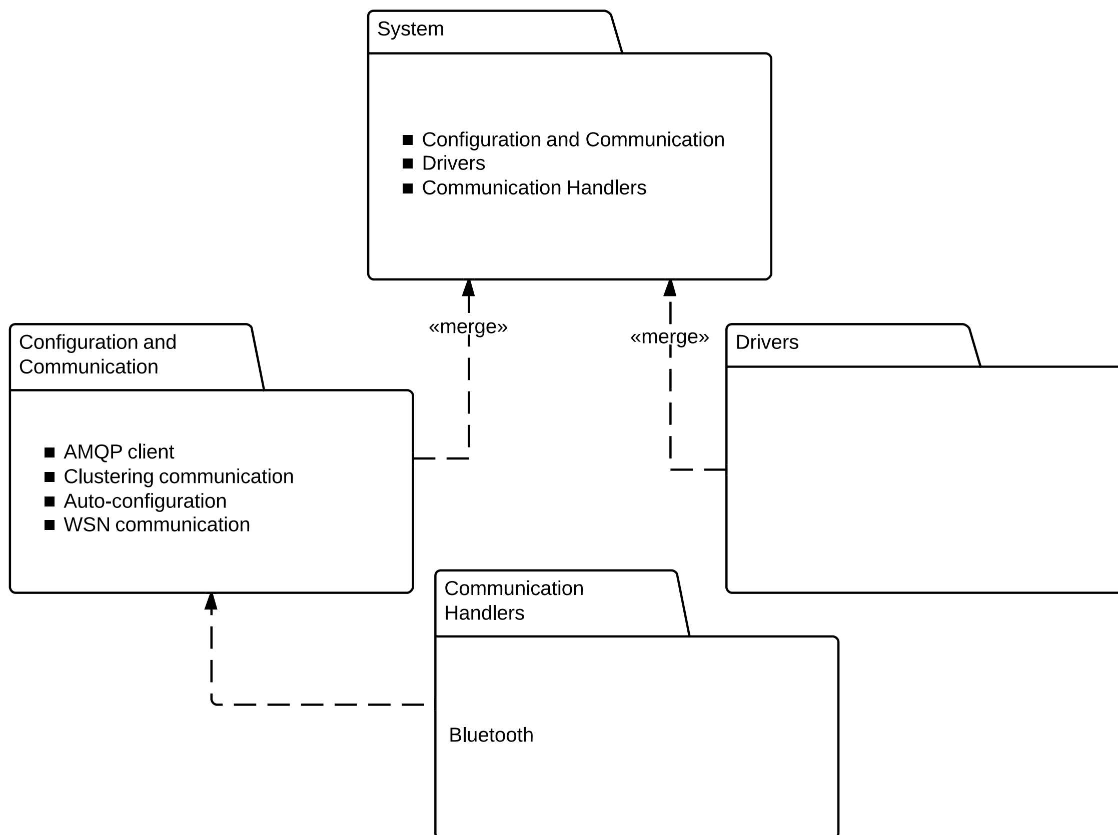

Figure 4.4: Wireless Sensor Network node architecture.

4.4 Implementation Details

In this section, the WSN node architecture will be explained in more detail, focusing in the actor hierarchy and explaining each feature properly.

4.4.1 Concurrency and fault-tolerance

Building an open an expansible framework for a WSN could lead to a more error-prone system, since there is no full control about third-party software's behavior. As stated above, there will be several virtual services running on the top of each node. Given the dynamic nature of the system, services have concurrent tasks such as collecting data from a sensor and publishing it either in a local service and to the *cloud*. In order to solve this problem, the *Actor model* was adopted, decoupling each service from the main process, ensuring that if it is experiencing issues, it can be restarted or stopped permanently without compromising the rest of the system. As an example, if a service such as a web service that process collected data crashes it will be possible to restart it without any technical intervention or system breakdown.

4.4.1.1 Akka

Akka[11] is an open-source framework which goal is to simplify the construction of concurrent applications on the Java platform, using the Actor Model. Akka is written in Scala and it is available both for Scala and Java. The actor hierarchy used is presented in the Section 4.4.2.

4.4.2 Actor hierarchy

The systems' actor hierarchy is shown in figure 4.5. It is separated in two main components:

Device communication

This component is responsible to discover new Bluetooth devices to connect and for starting the communication with them, starting the respective drivers and opening input and output streams. Furthermore, this component is responsible to forward messages to an AMQP server hosted on the *cloud*. The actors of this module are described below:

Bluetooth device discovery which is responsible to find the available Bluetooth devices as well as the available services provided by them.

Cloud forwarder which is responsible to forward the information sent to the *cloud* to the proper service, properly configured with the URL, username and password when the application starts.

Connection reader which is instantiated for each open connection, performing all the read operations.

Connection writer which is instantiated for each open connection, performing all the write operations.

Driver which is responsible for the communication with each driver, calling the initial setup and redirecting the read and write requests to it.

Cluster

Discover peer nodes and cluster with them is another application's core functionality. The nodes share a message queue in order to communicate within each other. There are two actors associated with this features:

AMQP subscriber which is responsible to subscribe a message queue from the RabbitMQ cluster, allowing to exchange information and serialized tasks across the peers.

Peer discovery which is responsible to send a multi-cast message with the information of the local RabbitMQ service. Also, it listens the multi-cast information announced by other peers, in order to establish the peer cluster.

[11] Akka's official website: `http://akka.io`

Both connection writer and reader rely on a dependency injection pattern, allowing their instantiation with any input or output stream. This means that the system only needs to know that any connection will have an input and output stream as well as way to connect (e.g., Bluetooth, ZigBee, etc), as shown in Listing 4.4. Every connection type must implement the *GenericConnection* trait, allowing to extend the types of connection without changing the framework.

```
1  trait GenericConnection {
2    def openInputStream(): InputStream
3    def openOutputStream(): OutputStream
4    def connect(driver: DriverDetails, context: ActorContext, log:
         LoggingAdapter): ActorRef
5  }
6
7  class BluetoothConnection(c: StreamConnection) extends GenericConnection {
8    val streamConnection: StreamConnection = c
9
10   def openInputStream(): InputStream = {
11     streamConnection.openDataInputStream()
12   }
13
14   def openOutputStream(): OutputStream = {
15     streamConnection.openDataOutputStream()
16   }
17
18   def connect(driver: DriverDetails, context: ActorContext, log:
         LoggingAdapter): ActorRef = {
19     val dev = RemoteDevice.getRemoteDevice(c)
20     val mac = dev.getBluetoothAddress
21     val actor = context.actorOf(Props(new Connection(this, driver)),
22       name = s"connection.${driver.driverName}.${UUID.randomUUID()}")
23     actor
24   }
25
26 }
```

Listing 4.4: Generic Connection trait and the Bluetooth connection implementation.

4.4.3 Interaction with the *cloud*

Published data is sent to the top layer of the global architecture (figure 4.3) with metadata that identifies the sensor from where the data was received, context information (e.g., user's identification) and the content itself. This information is required by the rest of the system to compare this data with the subscription list and understand where the data should be forwarded to. The input format published by the application, must be formatted using the JSON standard and include four fields:

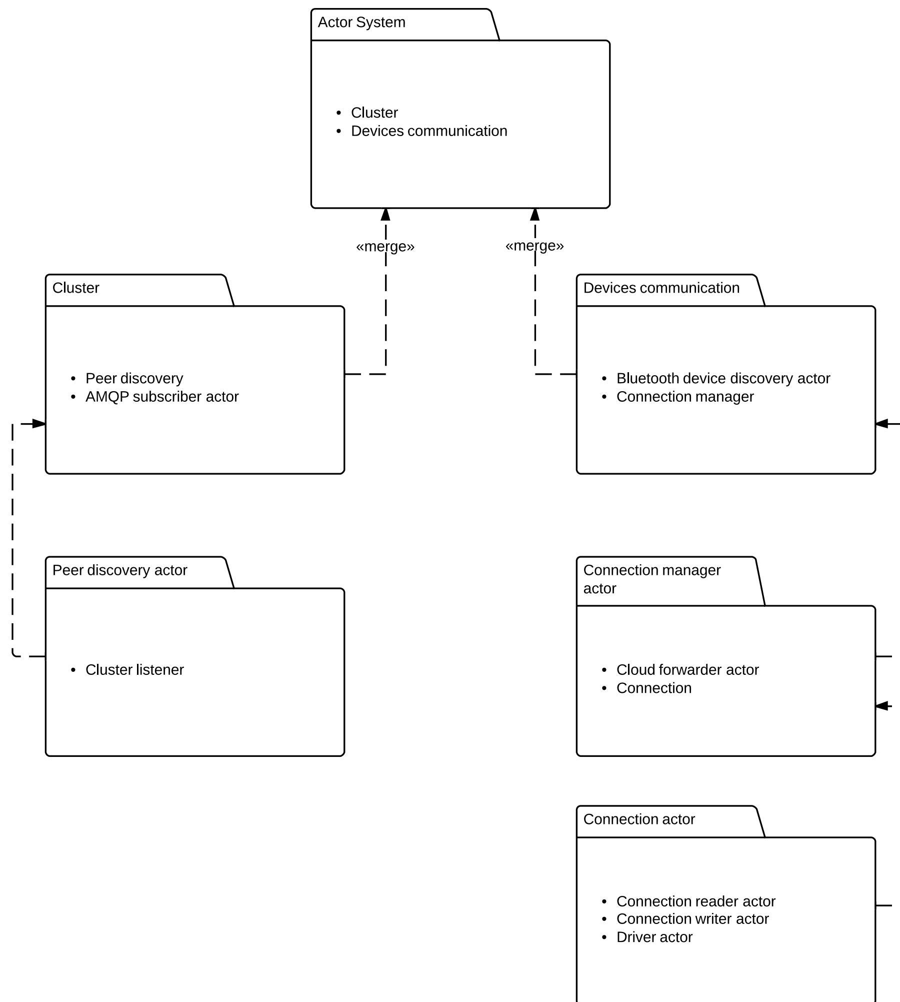

Figure 4.5: Actor hierarchy of the application.

origin
The unique key that identifies the sender and can be used to route traffic to it.

date
Unix time, corresponding to the number of seconds since 1 January 1970, of when the message was generated.

topic
Routing key, used to route the message to the proper subscriber(s).

payload
the content of the message, a JSON itself that might contain encrypted data for additional security. The final message must be a valid JSON-formatted string. As such, some characters are invalid in the payload (such as single " or \). If a message is not valid JSON, it will be

discarded on arrival. Moreover, the *cloud* service which receives the messages is content-agnostic, meaning that publishers can create their own messages, as long as they respect the JSON format. Listing 4.5 exemplifies how a message can be build.

```
{
  "origin": "user.patient1234",
  "date": "1388708123",
  "topic": "aalReminder.Reminder001",
  "payload":
  {
    "data": {
      "caretaker_name": "Dr. House",
      "patient_name": "John Doe",
      "prescription": [
      {
        "drug_name": "Brufen 600",
        "dosage": 1,
        "message": "Hello John. Please take your medicine",
      }
      ],
      "date": "2013-01-03 08:00:02"
    }
  }
}
```

Listing 4.5: A well formated message sent by the application to the *cloud*.

The messages are encapsulated in a Scala case class (see listing 4.6) and they are transformed from and to JSON using the Argonaut library.

```
implicit def NodeMessageJSON =
casecodec4( NodeMessage.apply,
      NodeMessage.unapply)("origin",
                 "date",
                 "topic",
                 "payload")

case class NodeMessage(origin: String,
             date: Long,
             topic: String,
             payload: String) {

```

```
13    def toJson = {
14      this.asJson.toString()
15    }
16
17  }
```

Listing 4.6: The NodeMessage case class and the implicit (de)coder from and to JSON.

4.4.4 Obtaining configurations

Configuration happens after a node is powered: the configurations are acquired from a remote service via HTTP to know how to set itself up. This is transparent for the user, as the node is previously registered in the system and knows to which context it must configure itself, downloading the appropriate drivers according to the list of service and drivers subscriptions from that user, available at the Component Management Server, as shown in figure 4.3. Although the development of a Component Management Server is not a requirement from this dissertation, it became necessary to build a prototype in order to assure the fulfillment of the auto-configuration requirement. When the application requests the configuration for the user, the response given by the server must be a JSON valid message with the following three fields:

version
: The version of the configuration file, as a date in the *yyyy-mm-dd* format.

akka_system
: The Akka system name where the system will be configured.

devices
: A list of the devices installed in the user's home, with the MAC address, port to connect and driver to download and use.

pub
: A list of services to publish data to the *cloud*, with the service identifier, device MAC address and the routing key where to publish data.

sub
: A list of services to subscribe data from the *cloud*, with the service identifier, device MAC address and the routing which should be subscribed by the application.

bt_services
: A list of the services names available from the Bluetooth devices.

amqp_server
: The AMQP server URL and credentials to publish and subscribe data.

An example of this configuration file is shown in listing 4.7.

```
{
    "version": "2014.01.07",
    "akka_system": "AAL4ALLBeta",
    "devices": [
        {"mac": "0002723D8BD1", "port": 2, "driver": {"file": "
            generic.jar", "class": "aal4all.drivers.ActuatorDriver"}}
            ,
        {"mac": "000666056165", "port": 2, "driver": {"file": "fhp.
            jar", "class": "aal4all.drivers.fhp.Driver"}}
    ],
    "pub": [
        {
            "id": "aal4all.user1.heartbeat",
            "device": "000666056165",
            "key": "aal4all.services.chardiac"
        },
    ],
    "sub": [
        {
            "id": "aal4all.user1.light001",
            "device": "0002723D8BD1",
            "key": "aal4all.services.domotics.light"
        }
    ],
    "bt_services": [
        "eCALLYX_bt",
        "NULL-SPP"
    ],
    "amqp_server": {
        "url": "http://bruxelix.inescporto.pt:8080/publish/",
        "username": "test",
        "password": "test"
    }
}
```

Listing 4.7: A configuration file for a given user.

In the lower level, as soon as the list of existing devices becomes available, communication is attempted with each device from that user. This level is responsible for establishing and maintaining communication only, with every communication being then handled by the drivers at the middle

level.

4.4.5 Device drivers

Drivers are developed by equipment manufacturers, by implementing an existing abstract interface, which assures the integration in the node. As shown in figure 4.6, the driver interface uses the proxy design pattern.

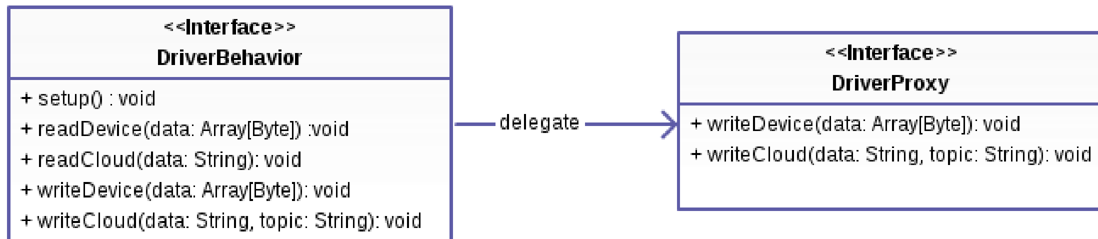

Figure 4.6: Driver Interface model.

Each manufacturer must implement the DriverInterface class, which has three methods: *setup*, *readDevice* and *readCloud*. The *setup* method can have actions which must be performed before starting the driver; *readDevice* and *readCloud* perform data reading from the device and *cloud*, respectively. The other two methods, *writeDevice* and *writeCloud* are delegated to the proxy class DriverProxy, and they are not accessible to the manufacturers. An example of the interface in Scala is provided in listing 4.8.

```
1  class RambleDriver(p: DriverProxy) extends DriverBehavior(p) {
2    /*
3     * Starts communication.
4     */
5    def setup() {
6      log.debug("RAMBLE DRIVER started!!")
7      // This behaviour may be extended
8    }
9
10   // Read from Device
11   def readDevice(data: Array[Byte]) {
12     Thread.sleep(1000)
13     writeDevice(UUID.randomUUID().toString.getBytes)
14   }
15
16   // Read from Cloud
17   def readCloud(data: String) {
18     Thread.sleep(1000)
19     writeCloud(UUID.randomUUID().toString, "test")
20   }
21 }
```

Listing 4.8: Driver Interface implemented in Scala.

Each driver is loaded using the reflexion feature of the Java Virtual Machine, which allows to load compiled classes into the running application, which in this case allows to load a given driver into the system without restarting it. As an example, listing 4.9 shows how drivers are dynamically loaded into the system.

```
1 val directory = "/drivers/"
2 val driver = "frunhofer.jar"
3 val loader = new java.net.URLClassLoader(Array(new File(directory).toURI.toURL),
      this.getClass.getClassLoader)
4 val externalClass = loader.loadClass(driver)
5 Some(externalClass.getDeclaredConstructor(classOf[DriverProxy]).newInstance(new
      ConnectionProxy).asInstanceOf[DriverBehavior])
```

Listing 4.9: Java Virtual Machine reflexion example in Scala.

Third-party drivers are a special concern in the presented solution. Despite all the testing that every driver must be submitted before be used by the system, it is possible that it fails. This fact must be taken into account and if a situation like this occurs it must not affect the all system. To solve this issue, it was implemented an actor supervision strategy, provided by Akka. The fault Handler takes as parameter a list of exceptions which will be handled, the maximum number of restart tries and within-time in milliseconds. An example is shown in listing 4.10.

```
1 val supervisorConfig = SupervisorConfig(OneForOneStrategy(List(classOf[Exception]),
       3, 1000), Nil)
2 val supervisor = Supervisor(supervisorConfig)
3
4 val driverActor = context.actorOf(Props(new DriverActor(d)), name = s"driver.${
      driver.driverName}")
5 supervisor.link(driverActor)
```

Listing 4.10: Driver actor supervision strategy

4.4.6 Clustering nodes

Acquiring data from sensors and triggering events to actuators is not the only functionality of the home node. In a multi-node deployed system, it is possible to balance the workload across the nodes, assigning tasks to the most available nodes and avoiding nodes overload, which could lead to crashes and data loss. This tasks can be loading hardware drivers, perform minor computation over the acquired data and propagate data to a higher level node, hosted on the *cloud*.

Therefore, the presented architecture allows decoupling and scalability, based on a multi-level node architecture. As an example, it is possible to have nodes which function is only to collect information from sensors and other ones running pre-processing and data aggregation services. In a superior level, third-party services may consume the processed data and show it to an user through a web interface.

4.4.6.1 A first approach

Using Akka as the actor model framework in which the system is built upon, it became an obvious first choice to use Akka-Cluster[12], an Akka extension to provide a fault-tolerant decentralized peer-to-peer based cluster membership service. However they were three main problems:

Inception
Creating a cluster with Akka-Cluster is an easy task. However, it was necessary to have one *seed* node which act as the *leader* of the cluster. Since we do not have control either on how many nodes will be deployed in a building and when they start the application, it was not possible to previously chose a *seed* node and inject it in a configuration file.

Electing the leader
Electing the leader of the system brought another problem. How could the leader be elected in a *blind* cluster (i.e., none of the existing nodes knew the others existence before) without race conditions? It was needed a third-party service, running outside the system in order to do that. The solution found involved the use of Apache Zookeeper[13] alongside with Apache Curator[14] library. Apache Zookeeper is an open-source centralized service which provides distributed configuration and synchronization service as well as a naming registry service for large distributed systems. Apache Curator is a Java library that simplifies the use of Apache Zookeeper, and it with a specific functionality to elect a leader between a set of candidates.

Partition recovery
In this kind of systems it is expected that one or more nodes can experience connection loss. This is because the nodes are meant to be spread all over a building and it is possible that, for instance, someone stumble on a node, unplugging it from the electrical power. Thus, when the node is plugged in again, it should start and join the cluster again. We concluded that Akka-Cluster have some issues with partition recovery, which make it unusable for our problem.

[12] Akka-Cluster official website: `http://doc.akka.io/docs/akka/snapshot/common/cluster.html#cluster`
[13] Apache Zookeeper's official website: `http://zookeeper.apache.org/`
[14] Apache Curator's official website: `http://curator.apache.org/`

4.4.6.2 A fault-tolerant cluster

As stated in the previous section, we concluded that Akka-Cluster is not ready for a production application. Therefore, the solution we developed relies in a multi-cast DNS system along with message queues in order to exchange information between peers.

Inception
When a new node is installed, the application should detect other peers in the same LAN and cluster with them. To achieve this, we used the JmDNS[15] library, which is a Java implementation of zero-configuration networking, a group of technologies that includes service discovery, address assignment, and hostname resolution, using multi-cast DNS records. There is a discovery actor per node responsible for detecting new peers and adding them to the list of the available peers.

RabbitMQ
RabbitMQ[16] is an open source message broker software that implements the AMQP. A distinctive characteristic is that RabbitMQ allows to easily build a cluster, replicating the message queues across the nodes. Moreover, RabbitMQ provides cluster partition handling out-of-the box, which is crucial on a fault-tolerant system.

To announce a service in the system, each node should register itself with the provided services, as shown in listing 4.11. After detecting a new peer in the system, each node should add the new RabbitMQ client, as shown in listing 4.12.

```
1  val serviceType = "_aal4all._tcp.local."
2  val serviceName = "inesc.gateway"
3  val port = 1268
4  val serviceProps = Map[String, String](
5  "name" -> "rabbit-node",
6  "address" -> "192.168.1.103"
7  )
8  val props = List[String]("name", "address")
9
10 val discovery = system.actorOf(Props(new ServiceDiscovery(serviceType, serviceName,
     port, serviceProps,props)))
```

Listing 4.11: Announce a service in the multicast DNS system.

```
1  val name = "rabbit2"
2  val address = "192.168.1.103"
3  Runtime.getRuntime().exec(s"rabbitmqctl -n rabbit join_cluster ${name}@${address}")
4  Runtime.getRuntime().exec(s"rabbitmqctl -n rabbit start_app")
```

[15] JmDNS offical website: http://jmdns.sourceforge.net/
[16] RabbitMQ's official website: http://www.rabbitmq.com/

Listing 4.12: Add a new node to the RabbitMQ cluster.

4.5 Adopting in AAL4ALL

AAL4ALL is meant to be an open ecosystem for AAL, allowing products and services development through certified partners. In this section there are shown a few particularities of the project and it is also explained why AAL4ALL is suitable for using the proposed architecture.

4.5.1 Low-cost computing

Multiple research projects in the AAL subject have failed to reach the market due to the unacceptable costs they would impose on patients or health-care institutions. While building a sensor network which should be deployed to most divisions of the patient's house, this aspect was taken into consideration.

In order to develop a prototype at an acceptable price, the Raspberry Pi computer was adopted. As shown in figure 4.7, it is a single-board computer built by the Raspberry Pi Foundation, initially intended to be used as part of computer science teaching in schools. The latest model is equipped with a 700MHz ARM processor, 512MB of memory, network card and it uses a SD Card for booting and long-term storage with its price set at $35 combined with a power consumption of only 3.5W. By plugging a Bluetooth dongle in one of the two available USB ports, it is provided the perfect device to build the nodes for the sensor network.

Although the application is prepared and tested to run on a Raspberry Pi, it is portable to any other architecture with support to the Java Virtual Machine.

Figure 4.7: Raspberry Pi model B.

4.5.1.1 Raspbian OS

Raspbian[17] is a Debian-based operative system. It was chosen to be the operative system deployed in the AAL4ALL nodes because it is optimized for the Raspberry Pi hardware and it has support to Debian packages generated as explained in Section 4.1.3, making easier to install, deploy and manage the AAL4ALL nodes. An image based on Raspbian OS was created in order to have the system ready to boot and cluster automatically.

4.5.2 Communicating with sensors and actuators

As mentioned in Section 3.1.3, Bluetooth protocol is low cost and have a low power consumption. These facts make it the preferred communication protocol by partners and general AAL research and production projects.

Communication can be performed in either direction, receiving and sending data to a remote device. Enhancing users' daily life is another objective of AAL4ALL and as such, devices should be able to perform actions and control mechanisms or systems to help the patient in his daily routine. These devices are called actuators. For instance, when a person with reduced mobility is in his bed and he remembered that the kitchen's light is turned on, he could turn off the light remotely, through a mobile application. Consequently, there is an emergent necessity of communication between sensors and actuators in a system like this. Although, commonly, the information flow of these devices is one-way only, i.e. they are either information publishers or subscribers, but not both. Using the WSN node as a communication bridge not only solves this issue but it also makes possible to exchange messages between different binary communication protocols.

4.5.3 System deployment

A possible and typical system's deployment at home is shown in Figure 4.8. Virtual services may consume data directly from the upper node (AAL4ALL server) because, usually, the applications run under a network-capable device such as a PC or set-top box. On the other hand, actuators have limited network capabilities, so they consume data via Bluetooth, subscribing topics at the home nodes. Each Raspberry Pi, representing a home node, will run two applications: the framework itself and the RabbitMQ service which allows to cluster all the available nodes.

Although this architecture was primarily thought to be deployed at the users' homes, it is possible to deploy it across a bigger building such as an health-care center, with several nodes across the divisions, allowing to capture data from several different patients without any change in the actual system.

Sensors

Two sensors: an Electrocardiogram sensor (ECG) and a blood pressure sensor (SpO2). These sensors communicate with one of the nodes in their communication range, sending them data.

[17] Raspbian's offical website: `http://www.raspbian.org/`

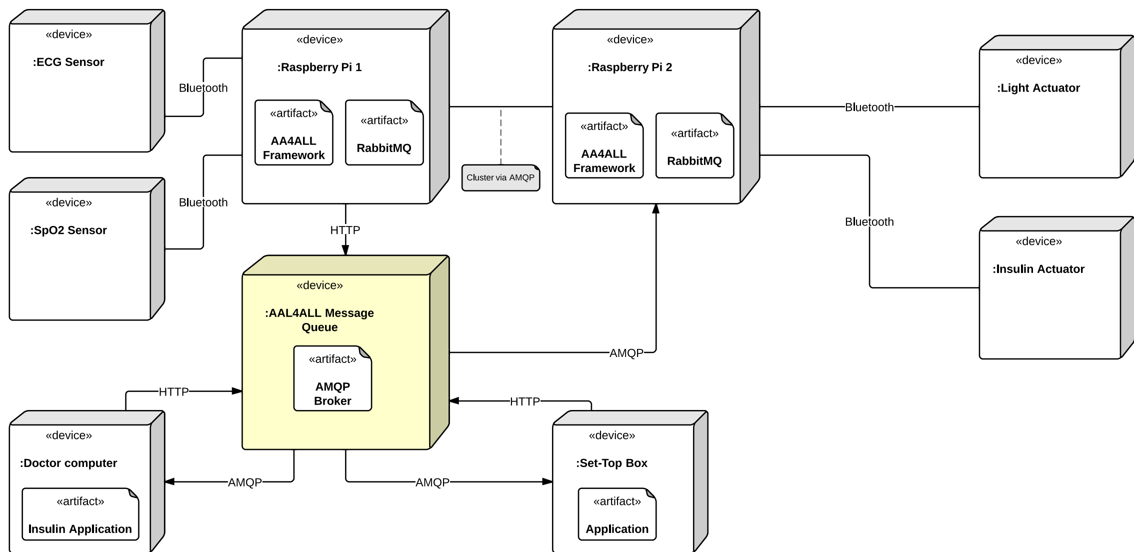

Figure 4.8: System deployment at home with two nodes in AAL4ALL.

Since the patient can move freely around the building, sensors might connect to different nodes in different moments.

Actuators

Two actuators: a light actuator which could be used to turn on/off the lights and an insulin actuator which could provide insulin doses to the patient upon request by the doctor's application.

Virtual services

Two virtual services: an application installed in the user's set-top box, to see his condition, videos and medical recommendations on TV and a an application installed in the doctor's computer, which allows to provide insulin to the patient, through the proper actuator.

4.6 Summary

The architecture was kept as generic as possible, so that it would adapt to the continuously expanding needs of this open ecosystem without requiring architectural changes. By doing so, portability between contexts was also achieved, with the proposed architecture being able to adapt to any other scenario where sensors and actuators need to communicate between each-other, as well as with could services. This happens not only at the sensor network level, but the whole architecture is able to orchestrate any domain where the *Publisher/Subscriber* pattern can be applied to propagate information. The involved entities can also be modeled as services which interact with the system.

An alternative application for the system would be a manufacturing process in a factory having sensors publishing data related to machinery production and actuators consuming such data. Thus items could be picked up from the output of a first machine and delivered to the next one as soon as it was ready in the production line. A monitoring service could raise alarms to the responsible

engineer when errors where detected and a mobile or web application could monitor the whole process so that managers could follow the production status. Moreover, in the same way, it is possible to apply these principles to domotics, ambient assisted living and other LAN monitor and control problems.

Chapter 5

Tests and Results

We believe that building an auto-configurable and self-managed system like the one discussed in Chapter 4 needs several kinds of tests to be proven as a reliable solution. This chapter describes the tests performed to ensure the fulfillment of this dissertation's goals.

5.1 Unit tests

A TDD approach was followed in this dissertation, so unit tests were written to all the relevant components of the system, ensuring that each component worked as an isolated element. The framework used for the test was ScalaTest[1] alongside with Akka-TestKit[2] in order to test the existing Actor system. An example of the performed tests is shown in listing 5.1.

```
1 "A ConnectionManager actor" should {
2     "close in a ClosedStreamException" in {
3       val actorRef = system.actorOf(Props(new ConnectionManager(new AMQPServer("
            http://%s:8080/publish/".format("bruxelix.inescporto.pt"),
4         "test", "test"))))
5       val result = Await.result(actorRef ? ClosedStreamException, 1 seconds).
            asInstanceOf[String]
6       result must be ("ClosedStreamException")
7     }
8 }
```

Listing 5.1: Unit testing example with ScalaTest and Akka-TestKit.

5.2 Integration tests

Integration tests were also conducted during this dissertation. They consisted in put together the all system and verify that it behaved as expected. The list of the requirements to assess the correct

[1] ScalaTest's official website: http://www.scalatest.org/
[2] Akka-TestKit's official website: http://doc.akka.io/docs/akka/snapshot/scala/testing.html

behavior is stated below:

- Download the last version of the application and start it correctly
- Download all the proper hardware drivers
- Search peer nodes and if they exist, cluster with them
- Search available Bluetooth devices and establish a connection
- Receive a message from the Bluetooth device and forward it to an AMQP server in the *cloud*
- Verify if the message was successfully received

5.3 Real application tests

The tests described in the previous sections allowed to show that the system worked but to assess if the dissertation goals were fulfilled, we conducted some *real world* application simulations. Some of these tests were conducted at Fraunhofer AICOS Portugal[3]. There were two components tested in this phase:

Nodes
There were two nodes: a Raspberry Pi and another computer, both running the same version of the application.

Sensors
There were two sensors: a door sensor connected to an Arduino[4] microcontroller sending real data and a mock sensor connected to the computer sending randomly generated data.

5.3.1 Stress tests

In order to assess the system response to a bigger number of messages sent by the sensors, André Pereira from Fraunhofer tweaked the Arduino to sent data from the door sensor in an infinite loop, during approximately 10 minutes. They were received and forwarded more than two million messages without a single crash.

Figures 5.1 and 5.2 shows the message processing and forwarding latency both from mock and the real sensor from Fraunhofer. The application processed and forwarded more than 80% of the data received in less than 200ms and more than 90% in less than 300ms.

As seen in Figure 5.3, the time consumed is divided in three parts: from the reception to the driver, driver operations and forwarding message. The messages sent from the Fraunhofer sensor are 8 bytes length and the messages sent from the mock sensor are 16 bytes length. Despite

[3] Fraunhofer AICOS Portugal official website `http://www.fraunhofer.pt/en/fraunhofer_aicos/about_us.html`

[4] Arduino microcontroller official website: `http://arduino.cc/`

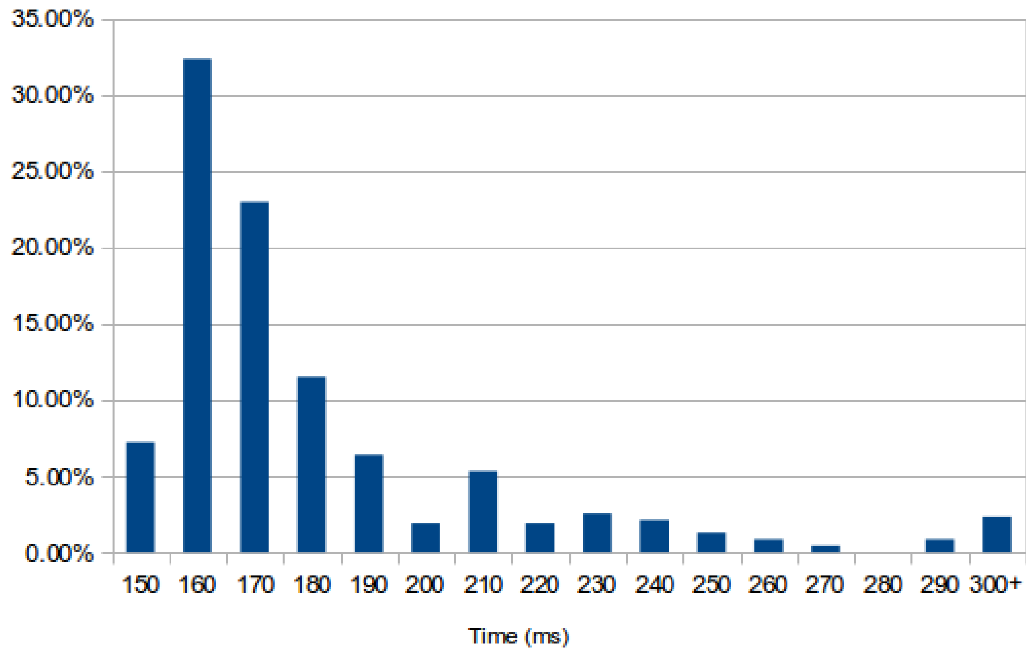

Figure 5.1: Message processing and forwarding latency from the mock sensor.

this fact does not affect the latency, the Fraunhofer driver executes a data checking algorithm, which consumes more 20ms on average to process the data. However, the time consumed passing messages between data reception and driver and forwarding messages to the *cloud* is similar, as seen in Table 5.1.

Table 5.1: Latency values across the three parts of processing and forwarding data.

	Received avg (ms)	Driver avg (ms)	Forwarding avg (ms)
Mock	0.301	0.383	177.34
Fraunhofer	0.309	20.2	177.75

Despite the lack of available sensors, there were performed more stress tests, in order to assess if the system was prepared for a larger data volume and for handling more sensors, like in a real use case.There were created 15 Bluetooth services in a mock sensor, sending data from `devrandom` as well as data from Twitter[5], using Twitter4J[6] library and a proper driver. The results of these tests were similar to the first, confirming the scalability of the system with more sensors and data volume.

[5]Twitter official website: `http://twitter.com`
[6]Twitter4J official website: `http://twitter4j.org/`

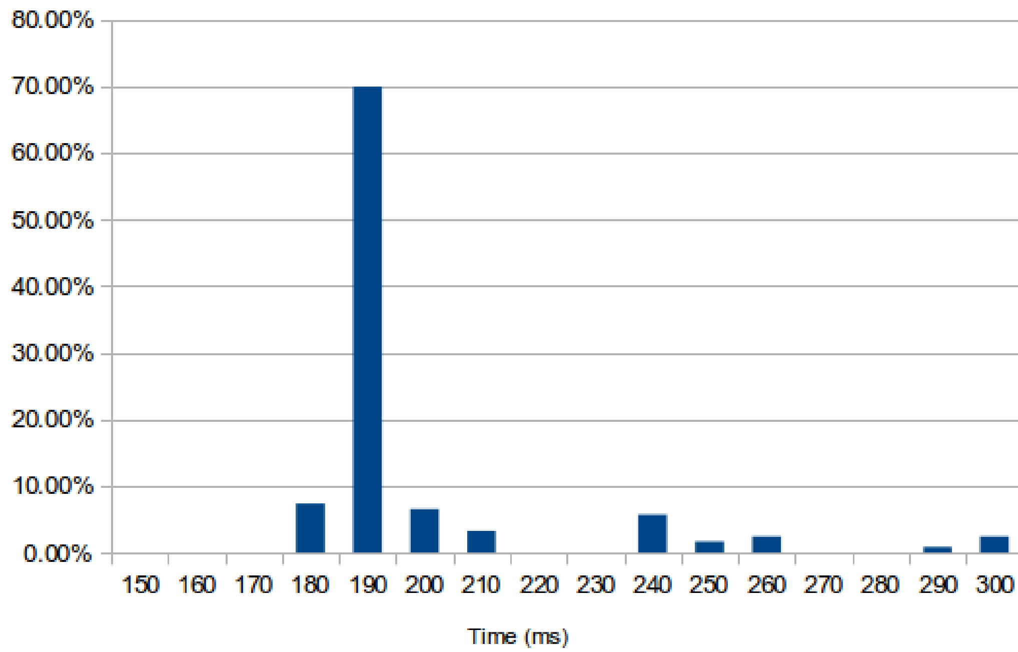

Figure 5.2: Message processing and forwarding latency from the Fraunhofer sensor.

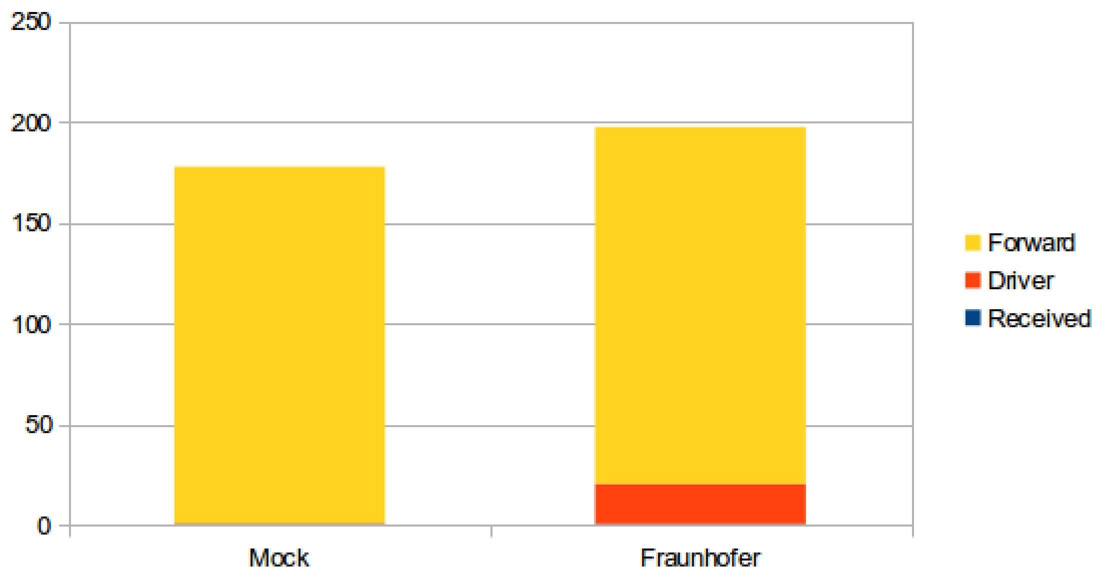

Figure 5.3: Difference in latency between a mock and the Fraunhofer sensors.

5.3.2 Reliability tests

To be proven as a reliable solution for this dissertation's problem, we decided to make two different tests:

- Load properly a third-party developed driver
- Restart one of the nodes's application and verify if it joins the cluster again
- Shut down the node where the sensors are connected to assess if they connect with the remaining node

Both tests were performed seamlessly. However, due to the use of Java Virtual Machine in a hardware with poor resources such as the Raspberry Pi, restarting a node take up to 40 seconds. About the third test, the connection between the remaining node and the sensors took up near 15 seconds each. In this case the mock sensor had not a *cache* mechanism, so the randomly generated data was lost. Nevertheless, the sensor developed by Fraunhofer together with their driver have a *cache* mechanism, so the real data was not lost and it was sent to the remaining node.

5.4 Summary

Following the need to validate our solution, we have performed different types of tests with the hardware available at the moment. Each separated component was testes isolated, as well as together in a system. Moreover, it was simulated a real environment, performing stress and reliability tests.

Based on the results and experience recollected during this dissertation, we perceived that some of the initial requirements are hard to achieve in this kind of auto-configurable, self-managed and fault-tolerant systems.

Chapter 6

Conclusions

This section presents an overview about the this dissertation's report, a summary of the contributions made and some guidelines about what can be done to extend and improve this work in the future.

6.1 Overview

In Chapter 1 we introduced the dissertation, presenting the context and explaining some concepts related with it such as AAL. We also explained the motivation of this dissertation and what were our goals.

Chapter 2 presents the problem addressed in the AAL4ALL context alongside with some concerns typically related with WSNs.

The literature about some background concepts of Wireless Sensor Networks is revisited in Chapter 3. Also there were analyzed four projects related with WSNs and AAL.

In Chapter 4 it was explained our methodology approach to the problem together with the implementation of the application details.

Finally, in Chapter 5 we described the performed tests as well as the gathered results.

6.2 Summary of the contributions

The three main goals defined to this dissertation (Section 1.2) were accomplished.

The research and work performed in this dissertation allowed to have a proof-of-concept framework to orchestrate and integrate devices in the AAL4ALL project. We kept the framework as generic as possible and we believe that the application built during this dissertation can be ported to other application domains, such as domotics, without a lot of effort.

Although the initial goals were accomplished, we experienced several difficulties building a fully decentralized, peer-to-peer and *blind* – i.e., none of the nodes knew about the existence of the others – cluster of nodes. These difficulties led us to abandon the Akka-Cluster solution and explore a solution based in multi-cast DNS and message queues.

From our tests we are able to conclude that the developed framework behaves as expected even in *stress* situations. However, in order to assess if the solution is ready for production, it is essential to perform more tests in a more real ambient with more devices and nodes. Moreover, the connection between the Component Management System and the framework must be redone as soon as the third-party entity responsible for it releases a specification.

6.3 Future work

During this dissertation, we gathered some ideas to extend and improve this work in order to release a production ready version.

Extend the connectivity
We developed our framework to work with Bluetooth enabled devices, because it was the only protocol consensual between the AAL4ALL partners. Most of them are still building prototypes and testing other technologies. This component can be extended in order to support other wireless protocols such as ZigBee or Z-Wave.

Create alerts on driver faults
There is a supervision strategy to handle third-party drivers failures. However, it would be useful to generate alerts in order to inform the system administrator of the failing driver.

Recollect data about the system
Since the developed system is meant to be deployed in several nodes across several buildings, it would be interesting to have access to data about the system (e.g., CPU usage, RAM consumption) in order to gather information to see, for instance, if is needed more powerful nodes than the existing ones.

Test the device limit
Due to devices limitation, it was not possible to assess the number of connections allowed by the Bluetooth library used. It would be helpful to test the framework with more different devices from different manufacturers.

Test in a real world situation
Although the all application passed through several levels of testing, when talking about software to monitor people's life it is necessary to test it in a real world scenario, monitoring the failures and infer if any data was eventually loss.

References

[AAL] AAL4ALL official website. `http://aal4all.org/`. Accessed on 30-12-2013.

[ABRV12] Alvaro Araujo, Javier Blesa, Elena Romero, and Daniel Villanueva. Security in cognitive wireless sensor networks. challenges and open problems. *EURASIP Journal on Wireless Communications and Networking*, 2012(1):48, 2012.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010.

[All07] OSGi Alliance. Osgi service platform, core specification, release 4, version 4.1. *OSGi Specification*, 2007.

[ANLR10] Cristina Alcaraz, Pablo Najera, Javier Lopez, and Rodrigo Roman. Wireless sensor networks and the internet of things: Do we need a complete integration? In *1st International Workshop on the Security of the Internet of Things (SecIoT'10)*, Tokyo (Japan), December 2010.

[AWW05] Ian F. Akyildiz, Xudong Wang, and Weilin Wang. Wireless mesh networks: a survey. *Computer Networks*, 47(4):445 – 487, 2005.

[BLHG02] Thomas Bodenheimer, Kate Lorig, Halsted Holman, and Kevin Grumbach. Patient self-management of chronic disease in primary care. *JAMA : the journal of the American Medical Association*, 288(19):2469–2475, November 2002.

[BMKK12] J.N. Bagale, J.P.T. Moore, A.D. Kheirkhahzadeh, and P. Komisarczuk. Comparison of messaging protocols for emerging wireless networks. In *New Technologies, Mobility and Security (NTMS), 2012 5th International Conference on*, pages 1–5, 2012.

[BMR+08] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *PATTERN-ORIENTED SOFTWARE ARCHITECTURE: A SYSTEM OF PATTERNS*. Number v. 1. Wiley India Pvt. Limited, 2008.

[BPC+07] Paolo Baronti, Prashant Pillai, Vince W.C. Chook, Stefano Chessa, Alberto Gotta, and Y. Fun Hu. Wireless sensor networks: A survey on the state of the art and the 802.15.4 and zigbee standards. *Computer Communications*, 30(7):1655 – 1695, 2007. <ce:title>Wired/Wireless Internet Communications</ce:title>.

[CCEF09] Marie Chan, Eric Campo, Daniel Estève, and Jean-Yves Fourniols. Smart homes — current features and future perspectives. *Maturitas*, 64(2):90 – 97, 2009.

[CMA04] L. M. Camarinha-Matos and H. Afsarmanesh. Telecare: Collaborative virtual elderly care support communities, in the. *Journal on Information Technology in Healthcare*, 2:73–86, 2004.

[CMRO04] L. M. Camarinha-Matos, João Rosas, and Ana-Inês Oliveira. A mobile agents platform for telecare and teleassistance. In Luis M. Camarinha-Matos, editor, *1st International Workshop on Tele-Care and Collaborative Virtual Communities in Elderly Care*, pages 37–48. INSTICC Press, 2004.

[CRMS09] Delphine Christin, Andreas Reinhardt, Parag Mogre, and Ralf Steinmetz. Wireless sensor networks and the internet of things: Selected challenges. In Technische Universität Hamburg-Harburg Institut für Telematik, editor, *Proceedings of the 8th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze, Hamburg, Germany*, pages 31–34, Aug 2009.

[DMSS12] Marin Dinu, Marius-Corneliu Marinaş, Cristian Socol, and Aura-Gabriela Socol. The impact of population aging on the sustainability of european social model. *Timisoara Journal of Economics*, 5(17):33–46, 2012.

[GH09] V.C. Gungor and G.P. Hancke. Industrial wireless sensor networks: Challenges, design principles, and technical approaches. *Industrial Electronics, IEEE Transactions on*, 56(10):4258–4265, 2009.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[HF08] Yang Hao and Robert Foster. Wireless body sensor networks for health-monitoring applications. *Physiological Measurement*, 29(11):R27, 2008.

[HKL+06] Tian He, Sudha Krishnamurthy, Liqian Luo, Ting Yan, Lin Gu, Radu Stoleru, Gang Zhou, Qing Cao, Pascal Vicaire, John A. Stankovic, Tarek F. Abdelzaher, Jonathan Hui, and Bruce Krogh. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. Sen. Netw.*, 2(1):1–38, February 2006.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS IX, pages 93–104, New York, NY, USA, 2000. ACM.

[HTSC08] U. Hunkeler, Hong Linh Truong, and A. Stanford-Clark. Mqtt-s — a publish/subscribe protocol for wireless sensor networks. In *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, pages 791–798, 2008.

[KJ03] S.E. Kern and D. Jaron. Healthcare technology, economics, and policy: an evolving balance. *Engineering in Medicine and Biology Magazine, IEEE*, 22(1):16–19, 2003.

[LBC+12] Xiaohui Liang, M. Barua, Le Chen, Rongxing Lu, Xuemin Shen, Xu Li, and H.Y. Luo. Enabling pervasive healthcare through continuous remote health monitoring. *Wireless Communications, IEEE*, 19(6):10–18, 2012.

[LSS07] Jin-Shyan Lee, Yu-Wei Su, and Chung-Chou Shen. A comparative study of wireless protocols: Bluetooth, uwb, zigbee, and wi-fi. In *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*, pages 46–51, 2007.

[MSPC12] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497 – 1516, 2012.

[OMSJ05] Chris Otto, Aleksandar Milenković, Corey Sanders, and Emil Jovanov. System architecture of a wireless body area sensor network for ubiquitous health monitoring. *J. Mob. Multimed.*, 1(4):307–326, January 2005.

[Org11] World Health Organization. *Global status report on noncommunicable diseases 2010 / [World Health Organization]*. World Health Organization Geneva, Switzerland, 2011.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[PB10] A. Pantelopoulos and N.G. Bourbakis. A survey on wearable sensor-based systems for health monitoring and prognosis. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 40(1):1–12, 2010.

[RMFJ+11] Artur Rocha, Angelo Martins, José Celso Freire Junior, Maged N Kamel Boulos, Manuel Escriche Vicente, Robert Feld, Pepijn van de Ven, John Nelson, Alan Bourke, Gearóid ÓLaighin, et al. Innovations in health care services: The caalyx system. *International Journal of Medical Informatics*, 2011.

[SGAV12] S. Singhal, A.K. Gankotiya, S. Agarwal, and T. Verma. An investigation of wireless sensor network: A distributed approach in smart environment. In *Advanced Computing Communication Technologies (ACCT), 2012 Second International Conference on*, pages 522–529, 2012.

[Tan02] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[Var07] Upkar Varshney. Pervasive healthcare and wireless health monitoring. *Mob. Netw. Appl.*, 12(2-3):113–127, March 2007.

[WAJR+05] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 108–120, 2005.

[Wil00] S.K. Williams. Irda: past, present and future. *Personal Communications, IEEE*, 7(1):11–19, 2000.

[WML95] S.K. Williams, I. Millar, and Hewlett-Packard Laboratories. *The IrDA Platform*. HP Laboratories technical report. Hewlett-Packard Laboratories, Technical Publications Department, 1995.

[WSO+10] Peter Wolf, Andreas Schmidt, Javier Parada Otte, Michael Klein, Sebastian Rollwage, Birgitta König-Ries, Torsten Dettborn, and Aygul Gabdulkhakova. openaal-the open source middleware for ambient-assisted living (aal). In *AALIANCE Conference, Malaga, Spain*, 2010.

[WVD+06] A. Wood, G. Virone, T. Doan, Q. Cao, L. Selavo, Y. Wu, L. Fang, Z. He, S. Lin, and J. Stankovic. Alarm-net: Wireless sensor networks for assisted-living and residential monitoring. Technical report, 2006.

[YIE11] M. Aykut Yigitel, Ozlem Durmaz Incel, and Cem Ersoy. Qos-aware {MAC} protocols for wireless sensor networks: A survey. *Computer Networks*, 55(8):1982 – 2004, 2011.

Appendix A

Twitter Driver

```

import twitter4j._

// Twitter OAuth config
val config = new twitter4j.conf.ConfigurationBuilder()
  .setOAuthConsumerKey("consumer key")
  .setOAuthConsumerSecret("consumer secret")
  .setOAuthAccessToken("access token")
  .setOAuthAccessTokenSecret("secret")
  .build

// Twitter Actor
class TwitterActor(t: TwitterDriver) extends Actor with ActorLogging {

  def receive = {
    case 'Start => start()
    case tweets: List[String] => tweets.foreach(t => .writeDevice(t))
  }

  def start() {

    val twitterStream = new TwitterStreamFactory(config).getInstance
    val listener = StatusListener(self)
    twitterStream.addListener(listener)
    twitterStream.sample

  }

}

// Status Listener. Sends 3000 tweets in a row
class StatusListener(parent: ActorRef) extends ServiceListener {

```

```
34   var tweets: List[String] = List()
35 
36   def onStatus(status: Status) {
37     if(tweets.size == 3000) {
38       parent ! tweets
39       tweets = List(status.getText)
40     }
41     else
42       tweets = tweets ++ List(status.getText)
43   }
44   def onDeletionNotice(statusDeletionNotice: StatusDeletionNotice) {}
45   def onTrackLimitationNotice(numberOfLimitedStatuses: Int) {}
46   def onException(ex: Exception) { ex.printStackTrace }
47   def onScrubGeo(arg0: Long, arg1: Long) {}
48   def onStallWarning(warning: StallWarning) {}
49 
50 }
51 
52 // Twitter Driver
53 class TwitterDriver(p: DriverProxy) extends DriverBehavior(p) {
54   /*
55    * Starts communication.
56    */
57   def setup() {
58     log.debug("Twitter driver started!")
59     val twitterActor = context.actorOf(Props(new TwitterActor(this)))
60     twitterActor ! 'Start
61   }
62 
63   // Read from Device
64   def readDevice(data: Array[Byte]) {}
65 
66   // Read from Cloud
67   def readCloud(data: String) {}
68 }
```

Listing A.1: TwitterDriver used to test the system.