UNIVERSIDADE FEDERAL FLUMINENSE

CARLOS EDUARDO PANTOJA

AN ARCHITECTURE TO SUPPORT THE INTEGRATION OF EMBEDDED MULTI-AGENT SYSTEMS

NITERÓI

2019

UNIVERSIDADE FEDERAL FLUMINENSE

CARLOS EDUARDO PANTOJA

AN ARCHITECTURE TO SUPPORT THE INTEGRATION OF EMBEDDED MULTI-AGENT SYSTEMS

Doctorate Thesis presented to the Programa de Pós-Graduação em Computação of the Universidade Federal Fluminense as requirement to obtain the Degree of Doctor in Computation. Area: Computer Systems

Advisor:
JOSÉ VITERBO

Co-advisor:
AMAL EL FALLAH SEGHROUCHNI

NITERÓI

2019

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

P198a Pantoja, Carlos Eduardo
An Architecture To Support the Integration of Embedded Multi-agent Systems / Carlos Eduardo Pantoja ; José Viterbo, orientador ; Amal El-Fallah Seghrouchni, coorientador. Niterói, 2019.
101 f. : il.

Tese (doutorado)-Universidade Federal Fluminense, Niterói, 2019.

DOI: http://dx.doi.org/10.22409/PGC.2019.d.05634242738

1. Computação Ubíqua. 2. Inteligência Artificial. 3. Sistemas Embarcados. 4. Produção intelectual. I. Viterbo, José, orientador. II. Seghrouchni, Amal El-Fallah, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

CARLOS EDUARDO PANTOJA

AN ARCHITECTURE TO SUPPORT THE INTEGRATION OF EMBEDDED MULTI-AGENT SYSTEMS

Doctoral Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense as requirement to obtain the degree of Doctor in Science. Topic area: Computer Systems.

Approved in September de 2019.

THESIS COMMITTEE

Prof. JOSÉ VITERBO FILHO - Advisor, UFF

Prof. AMAL EL-FALLAH SEGHROUCHNI - Co-advisor, Sorbonne

Prof. MARKUS ENDLER, PUC-RJ

Prof. ALEXANDRO SZTAJNBERG, UERJ

Prof. CÉLIO ALBUQUERQUE, UFF

Prof. ESTEBAN WALTER GONZALES CLUA, UFF

Niterói

2019

"And those who were seen dancing were judged insane
by those who could not hear the music"
Friedrich Nietzsche

Acknowledgements

First of all, I want to thank God for everything, and then my family for all the support provided during all these years. A special thanks for my girlfriend and life partner, Tielle, for dealing with all those hard times that we have been through. Other special thanks for my advisor, who taught me a different view of academic life and how to sail through those hard times. I also thank my co-advisor for welcoming me at LIP6 in Paris. I extend this acknowledgment to all of you who helped direct or indirect throughout these tough times. Thank you all.

Resumo

O desenvolvimento de sistemas onipresentes e difundidos considera o uso de componentes eletrônicos e sistemas de computador para aprimorar objetos diários com alguma inteligência computacional para ajudar os usuários em suas tarefas de maneira difundida. A Ambient Intelligence (AmI) é um ramo da computação onipresente que fornece um ambiente cheio de dispositivos interconectados e pode fornecer comunicação de dados e colaboração entre os dispositivos do sistema. Da mesma forma, a Internet das Coisas (IoT) fornece dispositivos ou itens identificados exclusivamente em uma rede para ajudar os usuários em suas atividades. Os sistemas multi-agente (MAS) são sistemas inteligentes em que os agentes são responsáveis por raciocinar, competir e usar recursos para alcançar metas desejáveis de forma proativa e autônoma. A abordagem do agente MAS é adequada para sistemas AmI, pois ambos compartilham as mesmas características. Nesse cenário, os agentes têm sido empregados em várias abordagens e trabalhos nos últimos anos. No entanto, nenhum deles considerou o MAS incorporado autônomo responsável pelos objetos IoT para ambientes abertos em um sistema AmI. Portanto, esta tese propõe uma arquitetura para o desenvolvimento de sistemas AmI que usam objetos IoT como coisas inteligentes com MAS incorporado para interface com sensores e atuadores e são capazes de se comunicar e interagir com outros objetos IoT em uma rede IoT. Em seguida, apresentamos a Internet das Coisas Inteligentes (IoST), que é uma arquitetura para gerenciamento de recursos, na qual vários Objetos IoT podem fazer parte de ambientes representados virtualmente e ficam disponíveis para serem acessados pelos usuários através do uso de aplicativos. A arquitetura é composta por três camadas que representam objetos da IoT, a solução em nuvem e aplicativos. Além disso, o IoST é estendido para criar uma camada dependente de recursos para que os agentes de planejamento contextual explorem os recursos físicos de qualquer objeto IoT disponível.

Palavras-chave: Sistemas Multi-agentes, sistemas embarcados, computação ubíqua

Abstract

The development of ubiquitous and pervasive systems considers the use of electronic components and computer systems for enhancing daily objects with some computational intelligence for aiding users in their tasks pervasively. Ambient Intelligence (AmI) is a branch of ubiquitous computing that provides an environment full of interconnected devices, and it can provide data communication and collaboration among system's devices. Similarly, the Internet of Things (IoT) provides uniquely identified devices or things in a network for helping users in their activities. Multi-Agent Systems (MAS) are intelligent systems where agents are responsible for reasoning, competing, and using resources to achieve desirable goals pro-actively and autonomously. The MAS agent approach is suitable for AmI systems since both of them share the same characteristics. In this scenario, agents have been employed in a lot of approaches and works during the last years. However, none of them considered autonomous embedded MAS responsible for IoT Objects for open environments in an AmI system. Hence, this thesis proposes an architecture for the development of AmI systems that uses IoT Objects as Smart Things with embedded MAS for interfacing with sensors and actuators and are able of communication and interacting with other IoT Objects in an IoT network. Then, we present the Internet of Smart Things (IoST), which is an architecture for resource management, where several IoT Objects can be part of environments that are represented virtually, and they become available to be accessed by users through the use of applications. The architecture is composed of three layers representing IoT Objects, the cloud solution, and applications. Besides, the IoST is extended to create a resource-dependent layer for contextual planning agents explore physical resources from any available IoT Object.

Keywords: multi-agent systems, embedded systems, ubiquitous computing

Figures

Tables

List of Abbreviations and Acronyms

AAL	:	Assisted Ambient Living;
AmI	:	Ambient Intelligence;
AOPL	:	Agent-oriented Programming Language;
AoT	:	Agent of Things;
BDI	:	Belief-Desire-Intention;
GW	:	Gateway;
IoT	:	Internet of Things;
IoST	:	Internet of Smart Things;
MAS	:	Multi-agent System;
MN	:	Mobile Node;
OMG	:	Object Management Group;
PRS	:	Practical Reasoning System;
SDDL	:	Scalable Data Distribution Layer;
WSN	:	Wireless Sensor Network;

Summary

Chapter 1

Introduction

Ubiquitous Computing or Pervasive Computing is the capability of embedding intelligence in everyday objects using computer systems to provide services, several functionalities, and information, continuously to support users in daily tasks reducing the level of interaction between users and devices or acting transparently in the environment [127]. The tendency of Ubiquitous Systems, supported by the advances in communication and network interconnection, allows to include everyday objects to interact with humans pervasively and to communicate with each other [116].

One of the sub-areas of Ubiquitous Computing is Ambient Intelligence (AmI), which comprises electronic and intelligent environments characterized by the interconnection of different hardware and telecommunication technologies to assist users' daily tasks autonomously and proactively [125]. Some issues about the development of AmI solutions refer to technologies that are necessary for the communication between devices in the system, collecting and storing context data and mechanisms for pro-actively acting on the environment. In general, the objects that participate in this system are composed of sensors and actuators. They are responsible for capturing information such as light intensity, temperature, and humidity. Besides, it controls specific functions in the environment such as lights, air-conditioners temperature, or a coffee machine, for example. In general, they are part of some embedded device, which is an electronic component with several limitations regarding hardware resources such as processing power and communication capability [111].

Early AmI works dealt with the automation of ordinary things at home or office. For example, the classical automated coffee pot [61] that sends information to the lab's persons about the making time, availability, and temperature of the coffee. This solution uses sensors and actuators, a computer, and a UNIX workstation to provide a simple

reactive solution connected to the Internet and without any intelligence applied in the solution. Another example is to provide available information and services about a room when the user enters it [45]. If users enter the kitchen, the refrigerator shows a list of fridge items or a list of recipes, for example, using Internet browsers. At that time, the projects did not consider communication between devices or embedded technologies for providing a more complex processing or autonomous reasoning, above all because of the technological limitations of that time. With the advent of IoT technology, it became possible to enhance and interconnect daily objects together to perform actions pervasively for helping humans being pervasively in different domains.

1.1 Internet of Things and AmI Systems

The Internet of Things (IoT) is a network of interconnected and uniquely identified devices, endowed with the abilities of computing and exchanging information without the intervention of human or computer interfaces. These devices work as everyday objects such as vehicle, refrigerators, televisions, smartphones, and every physical equipment, embedded with electronic components (microcontrollers, processors, sensors, actuators, and network connectivity) [4]. Some challenges of IoT deal with the heterogeneity of physical devices and to provide communication between interconnected devices.

Several fields and applications will be impacted by the Internet of Things (IoT), such as AmI. IoT can be used to deploy AmI systems since both use electronic components and communication infrastructure to provide such kind of intelligent environment. The IoT impacts the way of AmI systems are implemented since it allows an open architecture of devices that can be modified at runtime in contrast to design-time devices [105].

However, it is hard to deploy smart applications in existing IoT infrastructure for AmI systems because of the heterogeneity of devices that can enter in these systems. In an open infrastructure, devices can dynamically come or leave, and they should be able to exchange information by communicating with each other. Besides, the interoperability between these devices has also to be treated [76].

1.2 Agents and AmI Systems

Intelligent Agents are entities — constructed in both hardware and software — that are able of performing actions in specific environments autonomously and pro-actively based

on some cognitive model of reasoning. The Belief-Desire-Intention (BDI) [15] is a well-known and used model, which considers that agents can reason based on beliefs acquired from the interaction with other agents, environment or self-assertion to activate desires, intentions, and execute plans to achieve goals. A Multi-Agent System (MAS) is composed of intelligent agents capable of communicating and collaborating — or even competing — for using resources in an environment to achieve conflicting or common goals [130]. The use of the intelligent agents applied in AmI and pervasive systems are justified by the autonomous characteristics of agents and their application in complex systems, both found in Ubiquitous Computing and AmI [123, 68].

During the last decade, the agent approach was applied in AmI systems in several domains exploring emergent communications technologies (Wi-fi, ZigBee, Ethernet, etc.) in conjunction with Agent-Oriented Programming Languages (AOPL) and agent frameworks [30, 60, 74]. Some of the applications were based on logic, or traditional programming languages such as Java [81, 70, 104], and they were bonded to a specific domain with limited hardware technologies. In the same way, several works try to provide cognitive reasoning for controlling embedded systems integrating AOPL and hardware devices. Some of them exploit existent architectures and middleware to facilitate the communication between hardware and the intelligent software [100, 126, 27]. Some works try to embed MAS into hardware platforms to provide autonomy to the platform [63, 90], or to use a central processing unit for controlling the platform from distance [112, 6].

In general, those works employ a one-to-one mapping, in which every device or service in a real environment is mapped to an agent using a centralized architecture [108, 5]. It also uses distributed architectures [14], where agents of each device can be embedded into hardware with limited processing power and use communication technologies for exchanging information with other agents in that device. Furthermore, some works employ hybrid architectures mixing centralized and distributed agents [33, 117]. They consider the set of agents and their devices as a MAS responsible for the whole environment, and no new agents can enter or leave the system limiting the growth of the system at runtime.

1.3 Problem Setting

In dynamic environments such as AmI systems based on IoT infrastructures, devices can enter or leave the system anytime, growing, or even reducing the hardware infrastructure available for deploying applications. In this case, the infrastructure is not fixed, and it can

grow whenever new devices enter the system in a plug-and-play way, for example. In fact, there will be a fixed part of the infrastructure — such as electrical facilities adapted for AmI — and a mobile part that can grow depending on upcoming devices. Besides, these devices will be responsible for controlling a part of the fixed and mobile infrastructure available.

Issues such as dynamic environments, context management, scalability, and robustness of the system have to be taken into account for an entire application [94]. For example, in an AmI system where a smart lamp and a luminosity sensor can communicate with each other and negotiate based on the user experience for providing a better luminosity experience, new devices such as presence sensors, temperature sensors, or even users' mobile phones can enter or leave the system. These new devices should be able to interact with each other exchanging information to cooperate and achieve a collective goal. Besides, new devices can add new features to the original system. The AmI system must be prepared to guarantee that all services will remain working even if the number of devices grows, and the upcoming services provided by the new devices as well.

When deploying solutions for AmI systems based on devices, it is necessary to take into account that the system needs to be open and scalable allowing the interaction of the current devices with the new ones, which might be entering the environment. Another important issue is the configuration of a new device that wants to be part of the system. When considering AmI, this configuration has to be done without being perceptible by users regarding AmI's characteristics such as anticipative, adaptiveness, and pervasiveness. For example, in case of a smart lamp, as the environment is open, a presence sensor can enter to be part of the system, and communicate to the lamp to inform if there is someone in a room. In most of the cases, the user has to configure this new device manually.

These devices can employ different technologies and approaches on top of them to control their functionalities and sometimes to provide at least a minimum reasoning in their systems. This includes the agent approach itself. An embedded MAS can be part of an independent and autonomous device with enough processing power and memory for hosting a MAS. It is able of controlling sensors and actuators by using microcontrollers, serial communication interfaces, and tiny computers. The use of embedded systems is a critical issue to provide autonomy to devices in AmI and IoT systems and devices can be designed independently of the domain [91]. Therefore, applying MAS in embedded systems is not a simple task and problems can arise because of the number of percepti-

ons coming from sensors that a platform has access in real environments and real-time constraints that an embedded agent has to consider in response to a stimulus (acting in the environment). Some approaches try to deal with them [113, 89, 14] to help to create autonomous embedded MAS.

However, these works consider a MAS embedded in a device without the ability to communicate with other devices. Every agent present in the MAS is only able of interacting with agents from its own MAS and for controlling hardware components. Communication mechanisms must be present in such kind of devices considering IoT or AmI systems. If the embedded MAS does not have any mechanism for exchanging information with other devices, it will not be useful in a collective system where decisions have to be made considering distributed information. If the MAS could be deployed in a single and unique device able of interacting with other devices, some advantages can be exploited such as pro-active reasoning to discover context information in a real-time situation. Otherwise, using only one agent per device leads to an approach where the MAS has to be mandatory open, raising issues about privacy and security of the devices' infrastructure. Besides, depending on the agent's goals, it can be overloaded. In these cases, it is preferable to employ MAS in devices.

In general, AmI systems deal with many contexts produced from their devices and sensors. Context is defined as any information with some meaning used to characterize an entity in the system. An entity could be considered a person, place, or object [2]. These contexts are available to be consumed by other entities such as devices, systems, and services. In this scenario, devices will need to be continually being informed about context information or asking for it, which requires much effort when considering agents. Besides, there are no guarantees that the required device is still available in the system when someone tries to access it.

Agents have to work together to solve problems demanding collective efforts because resources such as knowledge and contextual information can be distributed in such systems. Sometimes, agents are not able to accomplish a goal, and it needs mechanisms to overcome this situation. When the agent does not have the know-how to do something, agents can solve problems by constructing plans and decomposing problems in subproblems. This planning mechanism help agents to formulate plans during its execution [46] based on distributed resources available, and as contextual information can be generated from devices in AmI systems, these data could be used as input for such planning mechanisms.

There is a planning management process named Contextual Planning [21, 22], which considers contextual information as part of the architecture of BDI agents for specifying new plans. However, this process focuses on producing a set of actions as a result that can be executed over an environment and it does not focus directly on how to handle those actions in devices or how to interact with devices to access contextual information in open systems. The absence of treatment in how to deal with these issues can lead to conflicts in concurrent accessing devices' resources since there will be several agents trying to request information or acting upon the environment.

1.4 Goals

The objective of this thesis is to present an architecture for the virtualization of devices and management of their associated resources — represented as sensors and actuators — to expose them to be consumed by clients in an IoT network. The architecture is responsible for keeping information about the real devices and resources stored in a middle layer, which abstracts the direct access to devices from clients. Hence, they can consume these data using applications, such as exposed web services, that access virtualized devices based on a Sensors-as-a-Service model. The Resource Management Architecture (RMA) is divided into three abstractions: Device Layer, Resource Management Layer (RML), and Application Layer. The Device layer comprises mobile and fixed devices running over an IoT network. These devices control sensors and actuators, hardware interface, and a tiny computer. In this layer, devices are also self-configurable and able to enter in the system for updating the RML with resources' data to be later consumed. Besides, IoT devices will be able to communicate with other IoT devices apart of the technology employed in them, considering the interoperability between those devices [114]. These devices are named as IoT Objects in this thesis.

IoT Objects can also employ embedded MAS for controlling all their resources, and to provide more intelligence, pro-activity, and autonomy to these objects. In this thesis, IoT Objects can be traditional Things from IoT or Smart Things employing embedded MAS. The overall architecture using Smart Things can be characterized as an Internet of Smart Things (IoST). In some cases, the use of embedded MAS bring some advantages compared to devices that only work as data repeaters sending information from sensors to a server application for generating context about a situation and need stimulus from other devices to act upon the environment [66]. Agents are pro-active, autonomous, and are capable of reasoning about information from the environment that they are situated.

These characteristics allow a piece of improved information or even a previous context generation before sending it to a server application releasing processing power of server applications, for example. Besides, agents can make decisions and act autonomously without depending on third-part processing (if they have sufficient processing power). For example, the MAS responsible for the smart lamp can communicate with the MAS responsible for the presence sensor to negotiate a better luminosity experience instead of sending this information continuously to a server-side for vertical reasoning and decision.

The RML deals with the virtualization, registering and updating of data coming from IoT Objects. It keeps a model of environments and their resources mapped in the core of the architecture that can be accessed by clients. The hardware information about IoT Objects is abstracted from clients, that need to access them by the interested environment or resources. Every time that an IoT Object sends data to this layer, it updates the core model keeping just the most recent values available. Besides, every action that needs to be performed by IoT Objects' actuators enters into a stack of actions to be executed in the respective IoT Object. The access to this layer occurs by using exposed services and web systems. The Application layer is composed of parts interested in accessing and consuming information over RML. It is possible for clients — systems, services, agents, or any other solution — to access these data and interact with virtualized IoT Objects, gathering information or sending actions commands. The RML abstracts the access of physical IoT Objects from clients, and it does not matter what kind of technology is accessing the virtualized data.

Particularly, when Contextual Planning agents are accessing resources, they need to verify the availability of resources, to gather a list of these resources and to allocate them for plans execution until they decide which plans are feasible and which resources will be involved in the execution. While an agent is reasoning with resources information, no other agent must take control of the resources involved in this process. Hence, all requested resources must be locked until agents decide which resources will be effectively used. After that, actions can be sent to be executed. However, Contextual Planning does not provide integration to physical devices or any sensors because the notion of resource is not defined in the Contextual Planning method. In this thesis, the RML is also responsible for checking the availability of the intended resources and for producing a list of available environments registered in the layer, and associated resources that can be used. It also deals with locking and unlocking process to guarantee the execution of an action effectively executed and a timeout process to avoid uninterrupted usage of resources. For example, when an agent needs to execute a plan, or even construct a

new one using contextual planning, to access an environment with an air-conditioner and projector, these resources should be blocked for other agents and services while it is using them.

The RMA layers are built using several different technologies. The IoT Objects are constructed using tiny computers, and they are controlled by an embedded system or MAS, which interfaces microcontrollers using a serial interface [63]. Both the embedded system and the RML use the ContextNet middleware [48], where IoT devices deploy the client part of ContextNet, and the RML deploy a server solution in the architecture. In the Application layer, web services and web pages are exposed using the Jetty servlet engine and Java.

Hence, we identify the following research questions to be tackled:

1. Can MAS be deployed for controlling a specific device embedded with sensors and actuators — working as a Smart Thing —, in a way that this device will be autonomous, proactive and independent from server side technologies to operate?
    - Can these Smart Things interact and communicate with other Smart Things for sharing information, plans, and goals.
    - Can any AOPL be used or extended to develop autonomous Smart Things since they use embedded MAS?
2. Is it possible to create an architecture to support the implementation of AmI systems, built over an IoT infrastructure, connecting IoT Objects and clients to provide consumable data as a service?
    - Can Smart Things self-register themselves at the architecture to send data continuously to be accessed by services and clients?
    - Could a service using the Contextual Planning mechanism use the architecture for accessing information from environments in AmI systems to verify the availability of resources, IoT Objects, and environments?
    - Is this architecture capable of dealing with matching, locking, allocation, and execution of physical resources for guaranteeing the fulfillment of contextual planning agents' requisitions?

1.5 Organization

This thesis is structured as follows: in Section 2, the theoretical background including all the basic concepts for the understanding of this proposal is presented; in Section 3, it is presented a review of the literature considering MAS architectures and works using the agent approach for IoT and AmI systems; In Section 4, it is presented the proposed architecture; In Section 5, it is presented the implementation of all components of the architecture; In Section 6.2 it is presented some performances tests and a case study. Some final remarks are presented in Section 7; and finally, the references are shown.

Chapter 2

Theoretical Background

In this chapter, it is briefly presented the basic concepts for helping to understand the goals of this thesis. Firstly, it is presented the agent and MAS fundamentals, where the main concepts about the openness of environments and systems are addressed. Then, it is shown some of the technologies for the development of MAS; afterward, we present the relation between middleware and the IoT focusing on the *ContextNet* middleware, which will be used to provide connectivity and communicability in this thesis.

2.1 Agents and Multi-Agent Systems

Agents are intelligent entities coming from Artificial Intelligence that is capable of sensing an environment where they are situated, and after a reasoning process, they are able of acting upon the same environment in order to achieve common or conflicting goals. Agents are autonomous entities capable of learning from their past experiences, interaction with other agents, and perceiving the environment. They are also adaptable and flexible, being capable of reacting pro-actively to changes within the environment they are situated. An agent can be part of a system where it can interact in a peer-to-peer way with other agents. Hence, a Multi-Agent System (MAS) is a system, which comprises several agents competing or collaborating for achieving individual or collective system's goals. Every agent of a MAS are responsible for acting and sensing a sphere of influence, which is a piece of the environment, and more than one agent can overlap this sphere, competing or working together based on their individual or collective goals [131].

These characteristics of agents are often related to AmI and Ubiquitous systems since they need intelligent, autonomous and pro-active entities capable of interacting and reason in a highly dynamic ambient where persons and artifacts share information and cohabit

pervasively [68]. As AmI systems and ubiquitous computing consider systems built upon real ambients using physical devices taking actions pervasively, agents must follow up these characteristics. Agents can be virtual or physical entities [75]. A virtual agent runs simulated on a computer using a virtual representation of an environment, and it does not support an interface with physical devices or hardware. Conversely, a physical agent interfaces with hardware devices such as actuators and sensors to interact with real-world environments. It is embodied in physical infrastructure, and it can run embedded into platforms, tiny computers, or any computational technology. A traditional vision of a physical agent is a robot [103].

At this point, it is essential to define that both environments and ambients represent a physical or simulated place where information and entities — artifacts, devices, persons, etc. — can exchange messages, be perceived, handled and modified by entities present in the local at a specific time. In AmI, Ubiquitous Computing, and Ambient Assisted Living (AAL), for example, ambient is a physical representation of someplace such as a room or house with expected behavior and equipped with pervasive technologies. It is not limited to physical places, and simulated ambients are often used in many solutions [30, 67, 1, 72]. However, it is expected to materialize ambients and devices when adopting AmI or Ubiquitous systems. When considering agents, the environment is a place containing artifacts that can be perceived and used by agents or groups of agents in a MAS for helping in accomplishing their goals. Depending on the cognitive model adopted by agents, the artifacts' data assume different names such as beliefs and percepts. Many domains use agents as a programming abstraction, and simulated environments are not necessarily used as a trustworthy representation of reality. In some cases, agents can interact directly with the real world [75, 112, 66].

The open nature of a real ambient allows the entry and exit of entities at any time. It is natural to think in a room where it is possible to enter or leave freely or to add new features such as a television or a sound system. However, in these systems, environments can be open or closed depending on their configuration and purpose [23]. An open environment works as a real ambient allowing new artifacts and agents to enter or leave it in any time. Conversely, a closed environment allows artifacts and agents that exist from the beginning. This latter implies that the closed environment must be simulated since it does not make sense that exists a real environment that is closed for new entities or, at less, it is not useful. In this thesis, we consider an agent's environment as an open and physical environment that represents a real spot or place where exists agents that can perceive and act upon this same environment controlling objects.

The fact of existing open environments is related to the notion of MAS. If an open environment exists where agents can enter and interact with other agents, sharing information, and collaborating or competing, it is possible to consider that these agents are part of a dynamic MAS. In this thesis, we employ MAS on top of objects, and only objects can enter or leave the environment, which avoids the existence of standalone agents running freely in the environment. Hence, it is necessary to expand the discussion of openness to the MAS level.

Similarly to environments, MAS can be closed or open, depending on how they treat the mobility of their agents. The conventional approaches consider that agents established in a MAS cannot move to another system, and it does not allow existing agents elsewhere to enter in its system. Then, in a closed MAS, there is an architectural limitation that avoids the mobility of agents. When considering the communication of these agents, commonly the MAS also limits this interaction to agents that are part of the same system. However, agents able to communicate with agents from different systems does not change the openness of the system. If a MAS has open communicability, it does not mean that it is prepared for the entry of new agents in the system. Instead, an open MAS allows agents to enter or leave the system at runtime, and there are no limitations in communication since they can perform direct communication in the destiny system, for example. Hence, the openness of MAS is defined by mobile agents, which are capable of moving from one system to another and stay as long it is interesting for them or until they accomplish their expected goals [62].

However, in practice, it is hard to define a physical border for limiting agents inside a MAS and programming languages are responsible for dealing with the relation agents and system. In the same way, it is common to observe a MAS using hybrids approaches considering agents, communication, and environments. In this thesis, we treat a MAS as a closed system where particular types of agents can use abilities to communicate with other agents from other systems hosted in IoT Objects. However, the technologies employed in the development of our approach allows agents to be movable and systems to be open, and it has been explored in parallel approaches [41, 38, 39].

As stated before, agents are intelligent entities capable of reasoning about statements perceived from an environment and social relations to accomplish goals. For this, agents adopt cognitive models of reasoning that tries to model their mental behavior. A cognitive model tries to understand or scientifically explain basic cognitive processes involving the brain in how to accomplish complex tasks as perceiving, learning, and decision making [16].

There is extensive literature considering cognitive models in the agent domain, but one is highlighted at this point since it is used in this thesis. The Belief-Desire-Intention (BDI) [15] considers the cognitive process of practical reasoning based on beliefs, desire, intentions, and plans. Agents can perceive the environment as pieces of information named Beliefs, which are used for triggering desires and intentions of an agent. Plans and actions materialize Desires and Intentions considering agents' beliefs and goals, which define when the agent commits to a desire, transforming it in intentions. The BDI has been widely used in the development of MAS during the past years, including IoT and AmI.

2.2 Development of MAS

During the last decades, agents emerged as a paradigm of Artificial Intelligence for solving distributed and decentralized problems in different domains, including IoT recently. The agent approach provides abstractions and mechanisms based on cognitive models that facilitate the development of intelligent, pro-active, collaborative, and dynamic systems. There are several AOPL and frameworks used in the development of MAS in the literature using different reasoning models such as BDI or merely reactive agents, for example. The Java programming language was used in the last years for creating three of the well-known Java-based frameworks: Jade [7], Jack [19], and Jason [11].

Jade is a reactive framework where agents are developed in a Java-like style. However, there is an extension using the BDI model named Jadex [95] that runs over Jade. The frameworks Jack and Jason use the BDI architecture alongside an interpreter of the Procedural Reasoning System for providing real-time reasoning systems. Besides, Jason works together with CArtAgO [98], which provides an abstraction of artifacts placed in environments that agents can interact with and Moise [55], which presents an extension for normative and organizational models in Jason. These three technologies together are known as the JaCaMo [9] framework.

When considering applications for IoT and AmI systems using MAS, none of these frameworks are prepared for interfacing hardware and communicate to an IoT network. Hence, if one of the mentioned frameworks would be employed in the development of IoT Objects, interface mechanisms, or middleware should be necessary. The Jade itself has been used in domains such as robotics and IoT but since its reactive nature and the absence of a proper abstraction for implementing agents — agents are programmed in

Java — it is not the most appropriated framework to be employed in autonomous and intelligent IoT Objects, which is expected some cognitive behavior. Even Jadex, which implements the BDI model, still depends on Jade since this extension runs on top of Jade. The Jack framework is proprietary software, which makes it difficult to access the source code or modify the agent's reasoning cycle, for example. The frameworks Jason or JaCaMo have a BDI reasoning cycle implemented in Java, and they are open and free platforms. Besides, they have several points of extensions that can be explored for creating agents with modified behaviors, and to add external technologies. Then, it is essential to understand Jason's internal structures that will be explored in this thesis for creating Things and Smart Things.

2.3 Framework Jason

The Jason is a *framework* for developing MAS using the cognitive model BDI [15] and has an interpreter for the BDI based agent-oriented programming language *AgentSpeak* [97] in Java language. The BDI contains three basic constructions: beliefs, desires, and intentions. Beliefs are information considered to be true by the agent, which can be internal, acquired by a relationship with other agents or by the environment's information. Desires represent the agent's motivation to perform a determined goal. Intentions are actions that the agent is compromised to execute. Moreover, the Practical Reasoning System allows agents to build a reasoning system at runtime to execute complex tasks [11]. Besides, they have plans composed of actions that are activated depending on beliefs on their belief bases.

Specifically, Jason's standard agent has a reasoning cycle responsible for processing all perceptions and beliefs to generate events, which activate plans and actions. It is important to understand the reasoning cycle of a standard agent because several extensions (including the ones described in this thesis) modify some of its characteristics to enhance specific kind of agents with new customized abilities. The reasoning cycle (Figure 2.1) of a standard agent is composed of the following steps:

- **Capturing Perceptions**: The agent captures the perceptions from a simulated environment where it is situated. In this environment, the agents can interact with virtual objects that may have information represented as perceptions. In Jason, the perceptions and beliefs are literals. It is important to remark that the original distribution of Jason does not have any access to real environments using sensors

or actuators.

- **The Belief Update Function**: This function updates the Belief Base using the captured perceptions of the environment, beliefs received from messages, and self-statements generated internally during plan execution. For each modification in the Belief Base, an event is generated. An event represents a consequence of something that an agent has to deal with to achieve its goals based on those new beliefs.
- **Checking and Selecting Messages**: The agent has a mailbox for receiving messages from other agents. It verifies at the beginning of each cycle if exists messages to be read. Then, it can select socially acceptable messages to be processed or ignore the one that is not acceptable. This step also generates events.
- **Event Selection**: In this step, an event is selected from a list of generated events of the previous steps.
- **Dealing with Plans**: when an event is selected, it retrieves all the relevant plans of the agent's plan library. After that, a verification is performed to identify which plans can be executed based on its current beliefs and perceptions, and a function selects only one plan to be executed.
- **Selecting Intentions**: a function is responsible for choosing one ready-to-use intention at a time to be executed.
- **Executing Actions**: Finally, an action of the selected plan is executed one at a time.

The Jason does not have the necessary technologies for implementing IoT Objects for working in IoT and AmI systems. It does not interface hardware, and it does not communicate with different MAS. It is only possible to create MAS that accesses simulated environments as said before, and the communicability is limited to agents inside the created MAS. Hence, it is essential to adapt Jason for creating the IoT Objects. There is an extension of Jason [90, 89] named ARGO that uses a serial hardware interface [63] for transferring sensors' values as perceptions directly for agents and receives agent's actions to activate actuators, which helps the creation of embedded MAS using microcontrollers. This extension will be used in this thesis as part of the creation of IoT Objects and will be explained in details in Chapter 5.1.

For applying IoT Objects controlled by agents in IoT, open environments are necessary and, consequently, communicability and connectivity. In this case, middleware for IoT

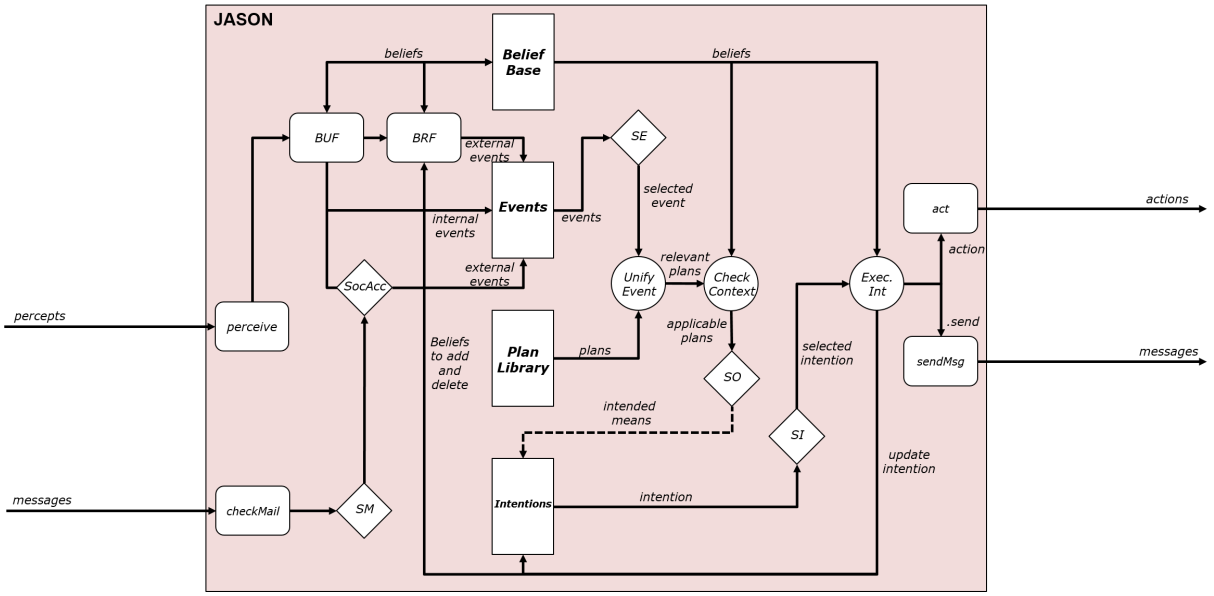

Figure 2.1: The reasoning cycle of a Jason agent [11].

should play an essential role in dealing with these issues since some of them can guarantee scalability, connectivity, communicability, data sharing, and protocols. In the next section, it is explored the chosen middleware, which is the basis for creating the proposed Internet of Smart Things (IoST) in this thesis.

2.4 Internet of Things

IoT middleware and protocols are responsible for managing the integration of the physical world and the cybernetic ones by using IoT objects and establishing an interconnected IoT network of these objects with the purpose of data collection and analysis, and reactive applications and systems [85]. The increasing number of middleware and connectivity protocols designed specifically for IoT do not consider the heterogeneity of such objects and their needs. When combining agent approach and IoT middleware for providing an IoST, the former one needs to be autonomous and pro-active and independent from the latter one. Besides, the IoT network should provide open environments where IoT Objects embedded with agents can enter and leave anytime they want.

Hence, it is necessary an open, lightweight and secure middleware capable of dealing with the heterogeneity of IoT Objects and agents technologies to be employed as the basis of the IoST without bounding the IoT Objects to the system. Besides, it is important to offer an IoT layer where different types of clients can access whenever it is necessary

without interfering in the IoT Objects functioning. In this thesis, we consider both IoT Objects and Clients as being built over agent methodologies.

A cloud-based IoT architecture should provide the necessary abstraction for creating an IoST capable of dealing with IoT Objects controlled by agents. Besides, it considers an uncoupled three-layer architecture where an IoT middleware working as a middle layer deals with connectivity and communicability of IoT Objects and clients. IoT Objects are capable of connecting and disconnecting from it as part of the device layer, and clients can interact or access these objects by accessing Application Programming Interfaces (API). Based on that, we selected the ContextNet middleware, which will provide all the necessary abstractions and constructions for creating the IoST. In the next section, the main concepts of ContextNet will be explained.

2.5 ContextNet

The *ContexNet* [47] middleware is a service for providing context data in stationary and mobile networks. It provides context services for ubiquitous and pervasive applications, and it has been employed in a wide range of solutions [36, 120, 49, 43]. It uses a Scalable Data Distribution Layer (SDDL), which employs the Data Distribution Service (DDS) [93] protocol for a real-time Publish/Subscribe mechanism for the communication within the SDDL Core. Besides, it also uses the Mobile Reliable UDP (MR-UDP) [109] for performing communication between mobile nodes and the core application running in servers. There are other services provided by the SDDL core, such as data persistence, data stream, fault tolerance, node disconnection, and group communication.

The data transferring occurs by using the MR-UDP and the Object Management Group (OMG) DDS. The MR-UDP treats messages between a client and a gateway, and the DDS is responsible for distributing data in the core of the network. The DDS is an OMG standard built upon a peer-to-peer architecture for data distribution, which guarantees Quality of Service (QoS) contracts between data users and providers. By using *ContextNet*, it is possible to enable the growth of a network, ensuring the scalability of the content distribution between millions of devices. From the point of view of the IoT developer, the ContextNet middleware allows the development of clients and core applications. Clients can be fixed or mobile devices able to connect to the server, and to communicate with other clients and to the core application. It deals with the MR-UDP connection to available gateways, and it isolates technical details from the application

layer. The core application deals with the data flow coming from clients, and it is useful for creating solutions where it is essential to collect data from several different clients and process this information somehow. In this thesis, the ContextNet middleware is used in the cloud solution as the core application dealing with all the requests from IoT Objects and other application that need to access to retrieve some action or act upon some environment. It offers all the necessary constructions to allow the development of the proposed architecture as the scalability, communicability, and connectivity.

Besides, the ContextNet also provides dynamic management for groups of clients. Devices running the client library of ContextNet can be arranged in groups, which helps to organize the collective goals and to facilitate communication since it uses broadcast and multicast messages. It is a promising characteristic to be explored when considering organizations of physical objects using MAS. However, in the scope of this thesis when considering the agent approach, ContextNet is used to create a new type of agent able to communicate with other entities, including other agents. This new agent will use a client instance of ContextNet, and it will be responsible for all inbound and outbound communications related to this MAS with other entities. As it is a ContextNet client, it will be uniquely identified in the IoT network. Hence, when this agent is embedded in Smart Things, it will identify this object uniquely in the network, providing the necessary connectivity and communicability for them to interact in open environments.

The Mobile Hub extends the capability of communication in ContextNet by running in mobile devices that adopt the Android operating system. It provides mobile communication and data processing extending the cloud SDDL middleware. It discovers nearby objects enhanced with wireless technology for short distance as Bluetooth, BLE, and NFC. As the Mobile Hub is a client from ContextNet, it is working as a bridge between these objects and the core of ContextNet, opportunistically connecting to those objects to transfer data to the cloud. Since Bluetooth has a distance range for objects to connect to the mobile device, we decided to employ wired connections to sensors and actuators, and we use embedded systems running ContextNet clients for managing these resources. In this way, it is possible to create Things and Smart Things capable of controlling resources that do not depend on distance since they are all connected in a single object. Besides, it is possible to adopt cognitive ability and reasoning in the case of Smart Things, which employs MAS.

Chapter 3

Related Work

During the last years, AmI systems have been applied in different domains such as Assisted Ambient Living (AAL), health care, smart homes, and smart cities, for example. The embedded systems and robotics domain also play an essential role in the development of this kind of systems because they are responsible for providing autonomy and pro-activity by embedding different technologies and systems in devices that are part of the environment. However, most of the early AmI systems were closed systems explicitly designed for an environment. Hence, new devices cannot enter or leave without the redesign of the entire system. Besides, the communication was restricted to all participants of the systems.

With the arrival of IoT technology, the devices could be connected to the Internet, and a device that is part of an environment was able of communicating with another one from a different environment situated in a distant local or place. The devices can be endowed with communication technologies, sensors and actuators, and capable of embedding software solutions. Then, considering the proximity between AmI and IoT concepts, it is reasonable using both in a system or to provide ways for two different systems to communicate using IoT technologies.

However, some authors argue that the current level of intelligence of devices are not sufficient for some applications because the devices act as data repeaters, transmitting gathered information from sensors to a server, which is responsible for reasoning and the decision making. These devices should be able to make their own decisions based on the perceived data from their sensors and act autonomously and based on social shares with other entities in the IoT network. At this point, one should state that the agent approach can support to provide a cognitive layer to such devices.

In this chapter, it is presented a review of the literature containing several related works applied in AmI systems and IoT supported by multi-agent approach during the last decades. It will be considered the initial work of AmI systems using the agent approach, and the evolution of them to employ MAS considering the IoT infrastructure. Firstly, it will be discussed early AmI systems using agents, considering the employed architecture, programming language, autonomy, and decentralized aspects of the system. Afterward, the discussion will be increased by the emergence of AOPL such as Jade, Jack, and Jason, and how they were applied in different domains for AmI. Some works considering embedded agent systems and robotic platforms will be discussed in how they could support the development of autonomous IoT Objects. Finally, it will be analyzed existent related works applying agents in IoT, considering all the relevant aspects mentioned before and also the use of IoT middleware, embedded agents, and MAS.

3.1 AmI Systems and MAS

Applying MAS in AmI is widely discussed in the literature, and several proposals integrating intelligent devices have already been presented during the last decades. In the beginning, the classical AmI systems were concerned about the automation of conventional furniture and electronic devices for aiding the resolution of simple daily tasks such as coffee machines automation [61], intelligent rooms [45], and refrigerators [84]. These projects used automation principles for providing some sensing and sometimes actuating aligned with the early Internet. Generally, they were ad-hoc solutions that did not provide complex reasoning or processing because of hardware limitations at the time. Similarly, embedded solutions or any communication technologies for providing interaction between devices were not simple to deploy.

A few years later, with the advance of the technology, it was possible to deploy MAS for AmI systems using traditional programming languages sometimes associated with logic in different domains. Mostly, these works presented a centralized multi-agent solution, where a central server is responsible for hosting the agents or a distributed multi-agent system where agents were distributed in a closed environment [76]. As stated in chapter 2, a closed environment is an electronic AmI system deployed in a specific facility, which it is not possible for new devices to be part of the environment at runtime without changing the electronic facility or redesign the software solution. A comparison of these related works can be seen in Table 3.1.

In the smart home domain, MavHome [30, 31, 29] is an intelligent house that works according to the necessities of the residents, learning about its inhabitants for offering comfort in day-by-day tasks, improving the inhabitants' free time, reducing operational costs, and managing security. The MavHome uses a MAS with hierarchical agents using an architecture structured in four layers named: decision, information, communication, and physical. The physical layer monitors the environment (using physical sensors), and it transfers the information to the Information layer using the Communication layer to transform raw data into context. Furthermore, the agent model is specific to the smart home developed, it was employed the Common Object Request Broker Architecture (CORBA) for the communication between agents, and the ZeroConf technology for dynamically deploy agents in the smart home model.

The fact of using a centralized architecture with agents controlling physical resources from a central computer increases the dependency of agents to the architecture and increases the risks of shutdowns of the whole system once that one single problem could stop the service. When using a decentralized architecture, it releases processing power from any central computer or processing, and consequently, the system becomes less dependent on these issues. Due to the autonomous and pro-active characteristics of agents, it is expected to explore them to manage physical resources locally, which helps in achieving the layers independence in the architecture. The use of agents for managing devices and resources was widely explored in the literature considering issues such as architectures, platforms, programming languages, organizations, middleware, and communication. We aim to combine these issues to provide an architecture where it is possible to extract the most of them.

Similarly, a system [18, 17] for a facility combines services with mobile and fixed devices using agents for providing continuous service for a group of persons in a museum. Every resource or device in the facility is modeled as agents with the capability of forming organizations. The agents communicate using an infrastructure named LoudVoice, which works broadcasting messages for all agents instead of peer-to-peer messages. The architecture is centralized with a one-to-one relation with agents and resources, which is not a problem when considering centralized reasoning. However, it could increase the number of agents in the system if the number of resources employed is high. Dealing with organizations and communication of agents in large MAS can be hard to deploy. The broadcast communication is costly to cognitive agents since they have to verify the proper addressee for every message and implement a discarding processing. At the same time, they are responsible for managing resources, which should be the main concern of

agents.

An AmI system and a test-bed named iDorm [51] uses embedded agents endowed with fuzzy logic applied in ubiquitous computing environments. The embedded agents are located on small computers (68000 Motorola processor) with limited processing resources responsible for receiving sensors information through the network. There is a network of physical sensors, and actuators in a room reproduced in virtual reality and controlled by a Java interface. There is a static agent responsible for the room and a mobile agent that communicates through wi-fi with the static agents. The solution provided is specific for the iDorm, and the agents do not control the sensors and actuators directly. Despite the use of a hierarchy of gateways for guaranteeing scalability, it is not possible for new devices to enter the system and iDorm's agents cannot communicate with other systems. Real ambients are naturally open, and it must allow new resources or devices to be allocated even if it is represented virtually. It is common to change some existing resources for recent ones or even to improve the ambient range by adding some new capability. Then, we comply with the AmI system's openness in our proposed architecture by allowing devices to come and go freely.

The SALSA [101, 102] architecture, in the domain of healthcare, aims to facilitate the development of AmI systems using autonomous agents. In SALSA, users, services, and legacy systems are represented as agents. Some agents are hosted on PDA devices, and the communication is provided by XML messages using an instant message server. As the PDA does not have enough processing power for delivering a fully embedded service using agents, it has to transfer information to the central part of the architecture. Embedded systems play an essential role in the development of devices with physical resources such as sensors and actuators. These systems deal in how data are acquired from sensors, and how actions can be performed. When the embedded system is an agent, it is responsible for managing the resources, and to reason about all information acquired. Depending on the amount of information available during these processes, it can affect the agent's performance significantly.

The Mobile-C [25, 26, 24] is an agent-based system for dynamic environments in the domain of real-time traffic detection. The system provides both mobile and stationary C/C++ agents, which interfaces hardware devices with hybrid control. The communication between agents is provided by a middle layer named Agent Communication Channel built on top of TCP/IP. Despite the hybrid hardware control, it adopts a centralized architecture for controlling resources. The hybrid capability of controlling different hard-

ware devices is essential when open environments are in question because it is expected that devices employing different technologies enter the system in anytime. The AmI system's architecture must be prepared to deal with this characteristic. The heterogeneity in devices could be understood as the employment of different resources — sensors and actuators— and platforms, or as using different microcontrollers where different resources can be connected. It helps in the adoption of a wide range of devices, and their designer could be free to choose the hardware that fit his project. In this thesis, we consider the microcontrollers heterogeneity as a functional requirement of IoT Objects in the proposed architecture.

There is a bookshop [108] where exists a one-to-one mapping between the system's elements and software agents running on a central computer. The system uses responsive ambients combining the agents and a physical environment. The architecture is generic, and there is a blackboard communication using Linda in Prolog. Afterward, the bookshop system is extended by using JADE, JavaSpaces, and Bluetooth for identification of shoppers and sensors [74]. Basically, there are several sensors in the bookshelves, which detects customers and their preferences. However, the solution is bounded to both domain and architecture. An architecture for dealing with devices in AmI systems should be capable of being adopted in any domain. Otherwise, every solution will have its own architecture, which can hinder the process of different interconnecting ambients, for example.

The ALZ-MAS [33] is a health care system for Alzheimer patients with five agents, which communicate with physical devices. The system uses RFID, wi-fi, network, and ZigBee. Agents running on a workstation processes the data. These agents are responsible for gathering information through RFID and control device using ZigBee. It also checks wi-fi for verifying if there is any PDA connected. The agents are hosted at the PDA or the workstation. The communication is using wi-fi and ZigBee. Case-Based reasoning and Java agents are used in ALZ-MAS and also in a Geriatric Ambient Intelligent system [32]. The architecture uses a MAS with BDI agents, where agents manager and patient run at a central server, and the nurse agent run on a mobile phone device. The ALZ-MAS is specific for the health care domain, and it uses an architecture where some agents are centralized. However, it leads to a distributed behavior where some individual agents are embedded in devices and communicate to the centralized ones. All the existing agents in the architecture compose the MAS. In our approach, we employ a MAS for controlling some devices connected to the architecture. Besides, it helps in separating devices from centralized agents or layers since agents from these devices are not related to any other part in the architecture.

An architecture integrates Service-Oriented Architecture (SOA) and intelligent agents for constructing AmI systems with functionalities divided by distributed services [118]. The core of the solution is composed of agents, working as controllers, and also modeled as services. A centralized architecture [5] based on a rule-based inference methodology using the Evidential Reasoning, where a single agent is responsible for gathering all information and decision making is presented in the smart home scenario. Depending on the number of resources that each agent has to deal with, it can overload the agent reasoning and delays may occur. In this case, MAS could be used to specialize and divide some agents obligations in devices.

An agent-based network for traffic management named aDAPTS [124] uses an architecture of three layers: a reasoning layer, middle layer for controlling, and the lowest layer for running agents. It describes both hardware and software implementation applied in a real-world scenario where agents are delivered for controlling centers, roadside controllers, sensing devices, and information systems using networks for communication. However, the solution considers agents as part of the same MAS, which limits the growth of the system at runtime, for example. In a car driving system [13], the system models the behavior of drivers using four distinct types of agents, which are instances of the Generic Agent Model (GAM). The entire AmI system is based on a single car, where the agents are responsible for the behavior of the driver, gathering sensors information, and artificial intelligence techniques. The car system is not able to communicate with another car, and it presents a simulated system.

A sensor and satellite tracking in cultural heritage applications using a MAS named Dalica [34] can infer users' interests from previous points of interest during a visitation in an archaeological area. There are three levels in Dalica system: a basic level providing external resources such as users' PDA, which interacts with the MAS, an abstraction level considering the authentication and the communication infrastructure and, an interaction-mediation level responsible for mediating relations between MAS and visitors' activities. Besides, sensors detect ambient data, which are transferred to agents using RFID, PDA, and network. The e-Commerce for shopping [59] presents an architecture based on a MAS with BDI agents. Each buyer and store have a personal mobile agent in mobile phones and PDA. It was developed using Java, Agent Factory, and it also uses Bluetooth.

In meanwhile, several frameworks for the development of MAS emerges. Frameworks such as Jade, 3APL, and Jack are being used in many domains, including AmI. Later, these same frameworks extend their concepts and approaches to include cognitive reasoning

Table 3.1: A comparison of the AmI related works.

Work	Domain	Language	Architecture	Platform
[18, 17]	Museum	LoudVoice	Centralized	-
[30, 31, 29]	Smart Home	CORBA	Centralized	-
[51]	Smart Home	Java + Fuzzy	Centralized	Mot68k
[101, 102]	Healthcare	Java + XML	Centralized	PDA
[25, 26]	Traffic	C/C++	Centralized	Hybrid
[108]	Bookstore	Linda	Centralized	-
[104]	Elderly	ALP	-	-
[33]	Healthcare	Java + CBR	Centralized	RFID+ZigBee
[118]	-	SOA	Centralized	-
[124]	Traffic	aDAPTS	Distributed	WSN
[5]	Smart Home	RIMER + RBI	Centralized	-
[32]	Healthcare	Java + CBR	Centralized	Mobile Phones
[59]	E-commerce	Java+Ag.Factory	Centralized	PDA
[34]	Tracking	Java	Centralized	PDA
[13]	Traffic	GAM	Centralized	Simulated

using the BDI model. The Jadex is one of those examples, which extended the original Jade distribution with cognitive constructions. Another framework also based on BDI agents, named Jason, emerges including an interpreter for AgentSpeak in Java. These days, Java-based agent frameworks play an essential role in the development of MAS because Java is still one of the most known languages for the development of software systems and components.

These initial AmI systems using MAS looks like nowadays applications available in smartphones that use sensors. These systems aid users by notifying about daily tasks even when the user is not aware of them. That way, the independent Lifestyle Assistant [60] monitors the behavior of health caregivers in emergency situations. The system was developed using JADE, and it provides agents for device's controllers, domain agents, planners, and system management in a server-centered approach. Besides, a group of agents is created for every human-assisted. The system was tested in real-world situations, and each agent had from four to seven sensors. As discussed before, the number of sensors can affect the performance of single agents significantly and generate undesired delays. In comparison with later works developed in the same period, these works just changed the programming language for an agent-oriented one. The centralized architecture is a common characteristic during this time, and several simulated systems employing agents were developed in a smart home domain.

There is a lightweight system for requesting cabs [81] that uses JADE-LEAP in PDA

devices, and it uses agents running at computers. The main idea is that the agent is responsible for requesting the cab using the PDA prototypes programmed with Jade. In these prototypes, there are agents associated with every PDA, that has to communicate with a centralized agent. The PDA agents are distributed in devices, but all other technologies are running centralized in a computer. In the same way, another work deals with one agent for every device, service, or content in a restaurant negotiator scenario [77], where every PDA is modeled as one or more agents, and the AmI system is composed of an ad-hoc wireless network for communication. An embedded approach in real-time using Jade for programming MAS for intelligent environments [58] maps each sensor and actuator to agents in the high-level language, and there are six types of possible agents. In fact, the number of agents would be the same as the number of sensors and actuators, and they are not embedded in sensors.

Some works apply the agent approach in simulated smart homes [28, 8, 65, 115, 71, 72, 3]. A MAS [65] with context-aware uses communication, interaction protocols, and the environment's information, where agents' task is to observe user's actions, predict and analyze the risk in executing the task. For this, sub-agents are receiving the simulated sensors' data, and transfer it to a super-agent connected with all sub-agents). However, connecting a super agent to all sub-agents can generate delays, and the solution does not provide an independent platform able to program MAS in different domains. These works address issues in how managing resources using agents in simulated homes. The interaction with simulated resources shows a direction that can help understand the behavior of agents when they have to deal with a lot of perceptions and data coming from sensors. Some BDI implementations have a costly process in reasoning with upcoming beliefs and perceptions. Hence, approaches for filtering perceptions or specializing agents can decrease the impact of this issue. Another interesting point is that these last three works use JaCaMo [9] (which Jason is part of) as the development platform, showing a direction of the use of the framework applied in real-world AmI systems.

3.2 Embedded Multi-agent Systems

In a scenario considering an AmI system, where real distributed and heterogeneous devices are responsible for managing an entire environment pervasively, where they can enter or leave this ambient in a scalable way, and devices should still act independently from the technological architecture employed, embedded systems can help to provide autonomy to devices regarding issues such as communication, embedded reasoning, and sensing and

Table 3.2: A comparison of AmI systems after the AOPL emergence.

Work	Domain	Language	Centralized	Platform
[60]	AAL	Jade	Centralized	Sensors
[81]	Vehicles	Jade-Leap	Centralized	PDA
[77]	Restaurant	Jade	Distributed	WSN
[74]	Bookshop	Jade	Centralized	Bluetooth
[58]	Smart Home	Jade	Centralized	Arduino
[28]	Smart Home	LABView	Simulated	Simulated
[65]	Smart Home	-	Simulated	Simulated
[8]	Smart Home	Jadex	Simulated	Simulated
[115]	Smart Home	Jade	Simulated	Simulated

acting in dynamic environments.

Robotics is a research area, which has been exploiting the use of embedded systems. Basically, robots are autonomous entities endowed with sensors and actuators, and some processing mechanism, including cognitive reasoning. Robot architectures deal with platforms, sensors, actuators, programming language, and reasoning mechanisms. One challenge is how to integrate these components in a way that a robot can deliberate to perform a task without failing to accomplish its goal. The concept of a robot and agent are quite similar, and several works try to provide cognitive reasoning for controlling robots integrating AOPL and hardware devices. Some of these approaches can be exploited for the development of embedded systems in IoT Objects supported by MAS for AmI systems in an IoT infrastructure.

Some of them exploit existent architectures and middleware to facilitate the communication between hardware and intelligent software. Some works try to embed the MAS into robotic platforms in order to provide real autonomy to the robot, and others use a central processing unit for controlling the robot from a distance. Applying MAS in robotics is not a simple task and problems can arise because of the number of perceptions that a robot has access in a real environment and some real-time constraints that a robotic agent has to consider in response to a stimulus (acting in the environment). These issues also have to be considered in IoT Objects in AmI systems. In Table 3.2, it is presented some related work, which uses AOPL and architectures for facilitating the development and construction of robotic agents.

There is a variant of 3APL [54] language for programming cognitive robots [35]. The 3APL architecture has constructions such as beliefs, goals, and actions. Besides, it has a deliberative cycle that is similar to the Practical Reasoning System (also used in Jason). The authors focused on the programming language, and they do not provide sufficient

Table 3.3: A comparison of embedded systems using AOPL.

Related Work	AOPL	Middleware	Platform	Hybrid
[35]	3APL	-	-	-
[56]	Jason	LeJOS	Lego	no
[80]	CArtAgO	Webots	-	no
[100]	Jadex	Stage/Player	Pioneer-2DX	no
[112]	Jade	LeJOS	Lego	no
[126]	GOAL	URBI	NAO	no
[6]	Jason	RxTx	Arduino	no
[121]	PANGEA	-	Arduino	no
[128, 78, 79]	Jason	ROS	Turtlebots	no
[27]	Teleo-R	ROS	Lego	no
[63]	Jason	Javino	Arduino	no
[89, 90]	ARGO	Javino	Arduino + Raspberry	no
[91]	ARGO	Javino	Arduino + Galileo	yes
[14]	ARGO	Javino	Arduino + Galileo	yes

information about the robot architecture and the communication between software agency and hardware devices.

A Jason extension for implementing Lego agents [56] programmed in Jason can communicate with the Lego Mindstorm NXT Kit using a middleware named LeJOS for integrating programs developed in Java and the Lego robot. The perception and the action function of the reasoning cycle of Jason were modified to allow agents to control robots using Lego. However, the communication between the Jason agent (hosted on a computer) and the Lego robotic agent is done using serial communication or Bluetooth connections. It does not truly embed the MAS, and it is dependent on the Lego Mindstorm Kit for mounting the robot. Despite the use of a MAS for controlling the robot, it is not possible to communicate with other Lego robots, limiting the approach when thinking in AmI systems.

CArtAgO [98] is used as a functional layer for providing artifacts that represent sensors and actuators of a robot, and Jason is used as the reasoning layer [80]. Despite using artifacts, which is an interesting abstraction for representing sensors and actuators in a MAS, because it provides a mechanism for agents to compete for resources and when one agent is using an artifact, it is blocked to anyone else. However, the authors use a simulator named Webots and do not embed the MAS. It is recommended for any approach deployed in real-world to consider interactions with physical resources to deliver proper management of AmI systems.

The Robot System Abstraction Layer is a generic middle layer for integrating MAS

and robots [100]. The authors evaluate the middle layer developing an example connecting a multi-robot system and a MAS using Jade. The middle layer takes significant participation in the reasoning through the use of behavioral components, and the communication between the middle layer and the robots are done using Stage/Player middleware, which is a network server for robot control. The Stage/Player uses TCP/IP connections to interact with sensors and actuators and can run in some manufactured robots. However, the Stage/Player does not communicate with microcontrollers directly, limiting the types of hardware that can be employed. Besides, the proposed layer is not scalable. In the same manner, an integration between Jade framework and Lego Mindstorm NXT uses the LeJOS as middleware to obtain Multi-agent Robot Systems with limited resources [112]. The agents of the MAS are running on different computers, which communicates with their respective robots using Bluetooth or serial port. Hence, the MAS is not embedded, and the robot was mounted exclusively using Lego robots and components.

A cognitive robot control architecture [126] using the AOPL GOAL [53] and a decoupled architecture separating the hardware and the cognitive reasoning of the robot is also presented. The URBI middleware is used for interfacing the hardware, and the communication between the cognitive layer is done using TCP/IP connections. The robot employed was the humanoid NAO, limiting the range of robotic applications in ubiquitous and AmI domain for example.

There is an unmanned ground vehicle controlled by a MAS hosted on a computer and programmed in Jason framework [6]. The agents are able to control the vehicle using external actions programmed in Java language, and the commands are sent to the hardware using radio transmitters on both sides (computer and microcontroller). A library for locomotion considering the eccentricity of the Earth supports the agent reasoning and positioning. The MAS is not embedded and depends on the radio transmitters for communicating with the prototype. Besides, it only communicates with a single type of microcontroller and a single board. In the same way, an AmI model supports residential accidents and emergencies using embedded agents with sensors and actuators, which use the Arduino and PANGEA as a platform for the development of intelligent agents [121]. For each Arduino, there is a specific controlling agent in the device and a high-level agent replicated in case the former is not capable of processing (because of limitations of Arduino).

Rason provides integration of Jason and the ROS middleware [96] for using high-level programming languages to program robotic agents. The integration is done by

modifying the original Jason distribution adding classes to embed Jason agents and ROS. An execution control layer is responsible for some sophisticated algorithms, and Jason is responsible for the decision-making process. The authors affirm that the perceptions burst affects the reasoning cycle of the agent significantly, which compromises its performance in dynamic environments. Similarly, some works [78, 128] focus in a scenario for fault diagnosis for mobile robots, and it uses Turtlebots as robots, and they propose the use of artifacts in CArtAgO.

An extension of Teleo-Reactive (TR) AOPL facilitates robotic implementations [27] where TR can be used for getting percepts from sensors using ROS as middleware, and robotic applications were developed using simulators and a Lego Mindstorm robot. A platform for embedding MAS programmed in Jason uses a Raspberry Pi board to control the basic functions of a ground vehicle prototype using external actions and middleware for exchanging messages using serial communication named Javino [63]. The agents are not able to directly control the devices becoming dependent on the simulated environment programmed in Java. It only communicates with a single type of microcontroller and several boards can be employed (but every agent's action for controlling the devices has to be programmed in Jason's simulated environment).

Another prototype of an autonomous vehicle assembled with Arduino in a real-world collision scenario uses ARGO agents [89] with its MAS embedded in a Raspberry Pi connected to an Arduino and four ultrasonic sensors, four temperature sensors and four luminosity sensors. The vehicle has to stop before it crashes into a wall. The use of ARGO sounds promising, and the experiments show that using a combination of ARGO abilities such as filtering perceptions could lead to acceptable performances in dynamic scenarios. However, it does not employ a heterogeneous hardware architecture, and it does not communicate with any other vehicle because there is no communication mechanism in the prototype.

A generic domain approach for prototyping ubiquitous and embedded MAS is presented in two different domains using ARGO: a vehicle and a smart home [91]. Both of them using two different types of hardware in the solution (it was employed Galileo gen 2 and Arduino). A prototype of smart bathroom [14] uses an embedded MAS for controlling several functions of a bathroom, including a safety functionality for users. The prototype evolves the previous work, extending the number of sensors and a complex scenario that is pervasively managed by embedded agents. The prototype uses an ATMEGA328 microcontroller and Raspberry Pi for embedding the MAS. However, these prototypes cannot

Table 3.4: A comparison of the IoT works employing agents.

Work	Domain	Language	Architecture	Platform
[64]	WSN	IntelHex	One Agent/Device	ATMEGAs
[106]	Semantic Web	Java+Jade	Centralized	Not Embedded
[82]	AoT	Conceptual	Conceptual	-
[99]	WSN	Protocols	One Agent/Device	Hybrid
[83]	AoT	SHOM	Conceptual	-
[107]	IoT	Jade	One Agent/Device	-
[52]	IoT	Eve+XML	One Agent/Device	Raspberry
[12]	Smart Grid	Javascript	Decentralized	Smartphone
[110]	IoT	Conceptual	Decentralized	Conceptual

communicate with other kind of vehicles or embedded devices, being considered a closed environment.

3.3 IoT and Agents

With the emergence of IoT technologies, it became possible to enhance and interconnect daily objects together in an open environment able to communicate over the Internet to perform actions pervasively for helping humans being in different domains. The use of the MAS approach in such a system occurs naturally because of the characteristics of agents. Both the IoT and AmI system deal with pro-active and autonomous devices capable of communication over a network to enhance an ambient with a certain degree of automation or even intelligence. There are several works in the literature, which uses agents to deal with some details of IoT such as collaboration, autonomy, and sense. A brief comparison is shown in Table 3.3.

Mobile agents provide collaboration in an IoT and Wireless Sensor Network (WSN) using an Application Programming Interface (API), and an architecture for providing interoperability between heterogeneous devices, platforms and WSN nodes [64]. The authors affirm that the composition model of agents facilitates the dynamic resource configuration of the system at runtime. The agent architecture is generic considering the programming language, and the system uses a table with some configurations that an agent should have. The agents are embedded in devices, and the API provided is responsible for allowing the interaction between devices and agent communication. In fact, there is at most one agent for each device during a period. The agents are also able to move from device to device using the HTTP protocol and from node to node in the WSN using CoAP protocol.

The authors provide a real prototype using smartphones and WSN nodes with embedded ATMEGA1284P and ATMEGA2560 microcontrollers. No AOPL is used in the solution (it uses IntelHex binary format), which makes it difficult for the agent community to adopt the approach. The use of different microcontrollers in WSN nodes are interesting since agents are unaware of them. The devices work as a spot where agents can move to perform some operations, then every node and device in this IoT network is not autonomously, depending on mobile agent occupancy. Besides, there is no mention of the scalability of the network.

There is a semantic web-based approach that deals with the coordination of several sensors, devices, and services in a ubiquitous environment for maximizing the user's satisfaction [106]. The agents are developed using an architecture named Coordination System of Real-World Transaction Services (CONSORTS), which can deal with the meaning of the resources present in a ubiquitous computing environment. CONSORT agents can manage devices, services, and users, coordinating all activities for greater user satisfaction. It employs semantic web agents that perform the reading of the environment's meta-data autonomously that can understand the knowledge related to this information.

The design principle of CONSORTS named *Physically Grounding* represents a real-world model with a spatiotemporal inference mechanism and a repository with sensors' and users' information. In the approach, agents are considered services, and they can communicate using FIPA-ACL. Wrapper agents are responsible for controlling devices using the same FIPA-ACL. The architecture CONSORTS was developed using Java and Jade, and it presents an application in museum domain where agents are not effectively embedded, and the whole ambient (including sensors, services, users and agents) are considered a MAS. Besides, it does not deal with the entrance of new devices such as sensors in an open environment or a communication mechanism between different devices, for example.

The Agent of Things (AoT) [82] is a definition for devices or things that are managed by a single agent in a ubiquitous computing and dynamic environments. The authors suggest that in such kind of environments, the approach used in most of the solutions to provide communication between devices are highly programmable and depends on a previous interaction configuration. It is proposed a conceptual framework which considers a direct physical interaction between IoT Objects using the hardware layer and using a software layer where every agent represents an IoT Object. In the conceptual framework, agents are situated and centralized in a software layer. Besides, there is no mention of

how an AoT will deal with scalability in real-world scenarios. The Software-Hardware Optimizer Model (SHOM) [83] identifies and analyses the integration of hardware and software for the creation of AoT working only at the design level.

The use of MAS in IoT raises questions about communication and the use of protocols such as CoAP, MQTT, and AMQP [99]. Hence, it proposes an architecture, where the main objective is to implement functionalities for controlling access in an agent-based IoT. The architecture uses a central server for controlling and coordinating communication using the protocols cited before. It is also presented with a hybrid system containing intelligent agents and IoT Objects in the traffic scenario in a WSN. There is a hierarchy between agents responsible for reasoning and communication with the embedded agents present in the IoT Objects. There is one embedded agent for each IoT Object, and the authors do not consider explicit how agents are organized.

The Agent-based Cooperating So (ACOSO) is used as IoT middleware for providing an IoT network where agents are devices, and the whole group of agents is a MAS [107]. The ACOSO supports the development of autonomous, interoperable, and cognitive MAS in the level of things. Hence, every smart object can be abstracted to a cooperating agent using Jade as AOPL. It provides a three-layered architecture: application; transport; and net and physical. The agents can managing sensors and actuators; reasoning and decision making using local and distributed databases; and a communication system for the interaction between smart objects and other mechanisms.

There is a decentralized approach with a framework and an architecture for engineering IoT applications based on autonomous smart objects [52]. The authors defend that traditional smart objects are data gathers and senders and the data is stored and processed in central servers compromising the autonomy of them, mostly because the high dependency on technologies to provide communication and reasoning in these systems. These characteristics lead users to deal with device configuration issues in the network. The proposed smart objects are capable of running even if remote technologies (i.e., gateways) are not available. There is one agent for a smart object, and it is programmed using XML technologies (it does not use AOPL). A prototype is presented using a web platform named Eve agent platform composed of Raspberry Pi model B and B+ and Arch Linux.

In the JavaScript Agent Machine (JAM) [12], agents can move from different JAM nodes, and the agent's behavior is modeled using dynamic Activity Transaction Graphs (ATG). In this architecture, agents are part of a simulated smart grid of sensors, and

tests were executed on a PC and smartphones. The architecture does not use any specific AOPL, and the work is simulated. The capability of moving from one spot to another can allow agents to move to another system to learn and exchange information. In the proposed architecture, agents are not forbidden to move from one Smart Thing to another. However, the mobility of agents is out of the scope of this thesis. Another work [110] proposes the idea of decentralized MAS responsible for cognitive intelligence in distributed computing for IoT directing the area for the development of genuinely autonomous smart objects that are independent of infrastructure technologies.

Chapter 4

The Proposed Approach

In this chapter, it is presented the Internet of Smart Things (IoST), our architecture for developing AmI systems using autonomous and embedded MAS as part of Smart Things that expose resources in the cloud to be consumed by services and clients. In Section 4.1, an overview of the architecture is presented. In Section 4.2 will be explained our proposal of Smart Thing, which is independent of the system's architecture to work and a MAS controls it; In Section 4.3, the Resource Management Layer and its components are presented; Finally, we present the IoST in Section 4.4.

4.1 Architecture Overview

In this section, we present an overview of the Resource Management Architecture (RMA), an architecture that supports the virtualization of devices that act as IoT Objects, exposing their sensors and actuators as resources to be accessed by clients that might be interested in their functionalities. Hence, these IoT objects can send data from their sensors to a cloud layer, and they also receive action commands to be executed in their actuators. The data is maintained in the cloud using a model that works as a mediator between IoT Objects and clients, isolating technical details from both of them and it organizes how the access to IoT objects' resources — sensors and actuators — is done. This cloud module is named the Resource Management Layer (RML) in this thesis.

Users can possess and share IoT Objects in a physical environment that can be exploited by other clients — as users or services — interested in their resources, for tests beds or any other purpose, or privately if the environment is confidential. Hence, some users do not possess IoT objects, and they can access and use some of the public resources available in an environment. As the architecture is not limited just to users, some clients

could be services as well. The RML provides exposed services that can be exploited by application developers and applications for accessing IoT Objects.

One of the available services is for agents in the Contextual Planning Layer (CPL) [21, 22], which need to use resources as part of their reasoning and planning mechanisms. These agents request the available resources, in a specific moment, to define the best plan that would fit the needed situation. In the next moment, the agents must have the assurance that when the plan is executing that the resources are still available for them. Then, the RML has an exposed service to deal with those requests and needs coming from contextual planning agents. Besides, users who own IoT Objects can manage their virtual environments using a web solution, and users that do not own any IoT Object can only visualize the publicly available ones using the web interface. The virtual environment is a computing representation of a real environment where IoT Objects should be related to facilitating their identification and localization in the AmI system.

Concerning the IoT Objects, they can be Things and Smart Things. The former one is the traditional Thing from the IoT, and the latter is an enhanced Thing empowered with a MAS responsible for adding more cognition to Things. Both of them share some of the functional and non-functional attributes (see Sec. 4.2) that are required to achieve the expected behavior in the architecture. They can connect in the RML to expose their resources for clients, which will interact with them eventually, as stated before. The Figure 4.1 shows the general structure and the organization of components in RMA.

From a bottom-up perspective of the architecture, we identified some issues to be tackled in this thesis. Things are capable of controlling resources — sensors, and actuators — in the architecture, but they have limited reasoning capability, autonomy, and pro-activity. For this, IoT Objects employing embedded MAS named Smart Things are introduced within the RMA. Using agents in IoT Objects, it is not a new approach and has been explored during the last years, as stated in Chapter 3. However, the use of embedded MAS for providing autonomy and independence to these Smart Things that can interact with other Smart Things through exchanging plans, beliefs, and goals at runtime with purposes of learning and coordination has not been explored.

Usually, architectures for the deployment of AmI solutions using IoT Objects are technology dependent. It means that the design of IoT Objects is homogeneous, using the same hardware and software technologies in their development. Besides, sometimes IoT Objects can be bounded the central layer of the AmI system. The RMA is a layered architecture that provides technology abstraction between its layers. Hence, it is possible

to mix both traditional Things, and Smart Things or any other technology using the Resource Management Layer (RML) because IoT Objects can run independently of the RML and the communication between them uses a text-based protocol. Bluetooth devices play an important role in such systems. However, the focus of this thesis is in IoT Objects capable of hosting embedded systems with wired connections to sensors and actuators.

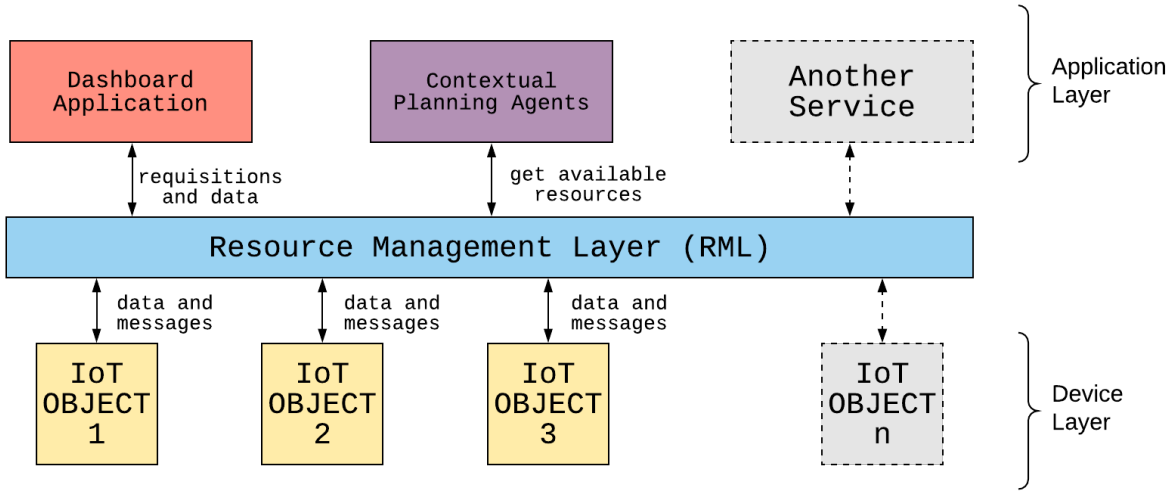

Figure 4.1: The three layer structure of the Resource Management Architecture (RMA).

The architecture provides low-coupled layers that separate concerns about IoT Objects, the RML, and applications used by clients. It means that technical details of IoT Objects are transparent for clients accessing the RML, and IoT Objects are unaware of who is accessing their resources. The RML plays a significant role in separating these concerns and dealing with requests from both layers. In this thesis, IoT Objects are built over considering the heterogeneity of hardware, which means they can employ different technologies — controllers, boards, sensors, and actuators — in the same structure, and different IoT Objects can employ different technologies as well. Hence, there is no need for all IoT Objects to use the same hardware configuration in the architecture. Logically, IoT Objects can also employ different technologies on top of them, including MAS controlling Smart Things.

The RML organizes the data received from IoT Objects and the requests for consuming data and using resources from clients, which access the cloud instance using available applications. These applications can be from different technologies as well since this access is done by exposed services that do not bound technologies used in applications and the RML. The architecture works providing the virtualization of IoT Objects, and it manages the availability of their resources by automating the registering process of IoT Objects

in the RML, dealing with actions requests coming from clients, organizing the available resources in environments that can be consulted, and using time-bounded functions to avoid unlimited use of resources. Agents can also be clients of this architecture if they need to make use of resources. Thus, when considering agent applications, the RMA is used by a contextual planning mechanism of agents for guaranteeing concurrent access to physical resources and the deployment of generated plans [20]. Besides, the architecture is expansible, and new approaches can be added as services, sharing its core model [88, 87].

Exposing devices as a service that can be virtually consumed brings some advantages since users do not need to have their own IoT Objects, reducing eventual costs in creating new ones. In this way, users can provide a shareable resource component that can be exploited for several purposes. The proposed architecture (Figure 4.1) is composed of three different layers:

1. **Device**: in this layer, an IoT Object is a device able of (i) connecting and registering in the Resource Management Layer when it starts. It is initially configured to be part of a specific environment, and it informs all of its available resources; (ii) it gathers data from all its resources, and it sends them to the RML and; (iii) it receives from the RML actions that must be executed by the IoT Object's actuators.

2. **Resource Management**: the RML is capable of maintaining updated information from IoT Objects hosted in environments and running over an IoT middleware. The Resource Management Component (RMC) manages (i) the registering process of IoT Objects; (ii) the resources' data updating process in the Virtualized Components Database (VCDB), and; (iii) the actions that must be executed by IoT Objects. Besides, it also capable of exposing services for providing access for clients to visualize environments and their resources, and for agents to access physical resources.

3. **Application**: it is responsible for applications that clients access to interact with IoT Objects and their environments, by visualizing data and sending action requests. In this thesis, it is represented as services to access environments' information, including contextual planning agents, which uses physical resources' information as part of their reasoning. This layer helps in obtaining data from real sensors, and it offers an interface for those who want to expose sensors as a service or to control some of them.

These three layers help organize the flow of information throughout the architecture,

and their main goal is to provide an uncoupled abstraction where IoT Objects are treated independently from the clients that may want access to them. The RML plays an essential role in managing resources and abstractions for both Application and Device layers since it is responsible for ensuring that they do not need to know technological details from any of them. The RML cares about how to address the right resource, which is part of an IoT Object, and it is placed in a given environment, to a specific client that desires to make use of it. For this, mechanisms for connecting at RML, data and IoT Object availability, the autonomy of IoT Objects, and heterogeneity of hardware and systems should be provided in each of these layers. The following sections will describe with details each one of these mechanisms for each RMA’s layer.

4.2 From “Thing” to “Smart Thing”

Things in an AmI system are entities responsible for monitoring the environment, sharing information, and communicating for aiding clients to achieve their goals. These Things are endowed with sensors and actuators — resources — and an embedded system for controlling its functionalities. However, they lack autonomy, pro-activity, and abilities for cooperating with other devices to offer a better experience from the perspective of the system’s users. These characteristics demand the use of technologies to provide certain autonomy and cognition in the decision-making process, including social exchanges with other entities. The agent approach, as discussed before, deals with autonomous, pro-active, adaptable, and social agents situated in the same environment where they can compete or work together for resources or achieving goals [131].

As stated in Chapter 3, there are numerous approaches, architectures, platforms, and prototypes applied in different domains that adopt agents for providing intelligence for their systems. Commonly, these approaches consider centralized reasoning where agents are running at a server solution representing services and devices in the system, which is not appropriated for our proposal. The system is highly dependent on the server, and there is no autonomy since they work as simple master-slave devices depending on agents situated outside the physical structure of the device. Following the evolution of the networks and microprocessors technology, agents started to be effectively embedded in devices providing the necessary autonomy to devices and systems started to consider open environments exploring characteristics such as scalability, connectivity, and communication. Yet, the approaches deal with the one-to-one relation between agents and devices, When comparing the performance of one or more agents responsible for a single

device, it is expected that later approach produces better results [86] once it is considered the specialization of tasks distributed between several agents compared to one agent responsible for all tasks.

Hence, in this thesis, we present the Smart Thing, an extended Thing from IoT managed by an embedded MAS capable of sensing and acting in an IoT based AmI System for aiding people or other systems to perform activities pervasively. Despite the practical employment of the agent approach in IoT, the main contribution is to consider an embedded MAS for controlling the functionalities of each Smart Thing providing autonomy and pro-activity for them. When employing a MAS as embedded system for a Thing instead of only one agent, the IoT Object should gain in performance and autonomy, since a unique agent can be overloaded depending on the tasks to be performed, even if the hardware technology is powerful enough to provide fast processing, parallel actions should still limit the agent performance. This parallel behavior of goals should affect somehow the Smart Things' autonomy since it is related to the capacity of agents to make decisions and act pro-actively in the environment. Smart Things endowed with a MAS have specialized agents performing different types of operations for orchestrating the internal and external activities — hardware interfacing and message exchanging, respectively — to deliver better performance of the Smart Thing including when occurring parallel goal that should be tackled.

This variety of specialized agents avoid Smart Things to be highly dependent on technologies that are not part of the Smart Thing's architecture. For example, the absence of a communication channel does not stop a Smart Thing of operating in any environment, because the agents responsible for interfacing hardware can operate even if the agents responsible for the communication do not reach any other Smart Thing. As stated before, several approaches rely on third part processing to work correctly. This characteristic makes the IoT Object highly dependent on a structure that is not part of its physical embodiment to work correctly, and this relation most of the time works in a master-slave way.

Besides, one of these specialized functions enables the MAS to communicate and interact with other Smart Things. Usually, MAS in such kind of Smart Things are closed systems — where agents cannot enter or leave to another system — and, consequently, no agents of a closed MAS can communicate to any other agent except those from its system. In Smart Thing, a specialized agent is responsible for all external communications with other Smart Things or MAS without a physical body. This characteristic leads to benefits

that can be exploited in the development of intelligent systems. First, these agents can exchange plans, beliefs, goals, and intentions with other Smart Things at runtime, which is a characteristic there is not performed by traditional Things. When a Thing is designed, its programming remains the same until the system or itself stops. Some approaches use rules that change the Thing behavior at runtime, but they do not learn new functions at runtime. When agents exchange plans, it is possible for a Smart Thing to learn new behaviors from another Smart Thing.

Traditionally, a Thing in IoT is a fixed or mobile device capable of sensing and interacting in an environment, and it is assembled with an embedded system, which is responsible for their functionalities by controlling existing sensors and actuators. The Smart Thing extends the notion of Things by applying the usage of MAS as the embedded system in such IoT Objects, allowing a MAS to make decisions based on their perceptions from the environment. A Smart Thing has specialized functions for each agent present in the system. The traditional approaches usually use one agent for each IoT Object, or it keeps the MAS centralized in one server machine, for example, interacting with the IoT Objects in a master-slave configuration. MAS brings more capability, pro-activity, and autonomy in executing a goal than only one agent responsible for all activities of an IoT Object.

Besides, raw data can be treated or analyzed before being sent to somewhere. In some Things, data is sent to a central solution, and after processing, it returns with some actions to be performed by it. Smart Things can deal with these requests without server interaction even if its necessary to obtain data from other IoT Objects once they are able to communicate with other IoT Objects in the same IoT environment. Of course, there are other solutions that try to do the same, but they do not employ an architecture that can exchange information with other IoT Objects or cognitive models for the decision making, for example. Their processing and operation rely on some technology informing them what they should do, and the whole system is bounded someway. One of the characteristics of Smart Things is to be autonomous and independent from any technology to perform actions in a system.

A traditional Thing in IoT shares some attributes (functional and non-functional) with the proposed Smart Things [122]. These concepts are based on the definition of functional and non-functional attributes from the object-oriented approach. However, based on the description of the Smart Thing, the following functional attributes can be identified and improved in some cases in order to provide real embedded devices that are able of acting in an AmI system:

- **Autonomy**: every Smart Thing hosts a MAS providing autonomy, and it should perform independently of any technology outside its architecture. The Smart Thing should still perform its functionalities into any environment even if there is no communication with other systems or IoT Objects. The mobile Smart Thing should work properly if it has been moved to another environment or AmI system.

- **Communicability**: since the Smart Thing is an autonomous entity, it should be able to communicate with other IoT Objects, including Smart Things themselves. Otherwise, it will not be able to exploit the advantages of the development of a network of Smart Things. Then, a communication infrastructure should be present in the Smart Thing architecture.

- **Connectivity**: Smart Things can connect at an IoT based infrastructure. Hence, a component for allowing connectivity in such kind of environments must be provided. In this case, middleware for IoT plays an important role because they are constructed to deal with several issues regarding connectivity and availability.

- **Context-awareness**: most devices act as data repeaters transmitting raw data from sensors to a central computer where data processing takes place. In Smart Things, this data processing can occur in the device without the obligation to send it to a central part. They can process and use the result for decision making, or they send this improved information to a server of the AmI system, which is not necessarily a central server.

- **Heterogeneity**: every Smart Thing has hardware components such as microcontrollers, actuators, and sensors that are responsible for the interaction between the Smart Thing and the real world. Hence, the Smart Thing should provide an uncoupled architecture for interfacing hardware components where a high-level programming language and more specifically, a MAS should be able of controlling these components. The type of hardware chosen should not interfere with the system responsible for logically controlling the Smart Thing. The heterogeneity of microcontrollers increases the range of possibilities for creating Smart Things. The use of sensors and actuators are considered heterogeneous since they are often connected to a microcontroller and does not matter what type it is.

- **Self-configuration**: Smart Things are capable of setting up itself in an AmI system using the RMA. When a Smart Thing enters in the system, it presents its functionalities. In contrast, the system informs the interesting parts of the presence of the

new IoT Object, and it provides an interface for interacting with those IoT Objects.

Besides, the following non-functional attributes are also identified in the proposed Smart Things:

- **Adaptability**: The Smart Thing must be applied in any domain. It means that they cannot be coupled to the deployed solution. For example, the technology for creating a Smart Thing for a smart home must be the same for creating one to traffic domain. Besides, a Smart Thing of a smart home domain could be able to move for an AmI system in traffic domain without any modification in its architecture or programming.
- **Computational Capacity**: since An embedded MAS controls smart Things, they must use platforms that are capable of storing and processing at least a complex MAS. A Smart Thing can employ notebooks, tablets, computers of any size, and any platform hosting an Operational System. In fact, there is not a size limit for Smart Things.
- **Interoperability**: independently from the agent language used on top of devices, it should be able to control Smart Things. Then, no matter what AOPL is being used for the development of Smart Things, they must be able to communicate with other IoT Objects, which do not have embedded MAS. In this case, they have to communicate using the same protocol. Hence, mechanisms for interoperability between systems must exist.
- **Reliability**: The Smart Thing must be reliable. It means that the hardware interfaces, reasoning, and all components of a Smart Thing should work properly as long as possible without crashing.
- **Scalability**: The Internet of Smart Things (IoST) must be scalable, allowing the entrance of new Smart Things without losing performance or crashing the system. Hence, every Smart Thing should be prepared to enter or leave the IoST.

Technologically speaking, a Smart Thing is equipped with microcontrollers, sensors, and actuators connected to any computational platform with sufficient processing power for hosting a MAS. Every Smart Thing is operated by an embedded MAS composed of agents with different purposes. There are agents responsible for interfacing with hardware devices; communicating with others IoT Objects; and traditional agents responsible

for reasoning and internal actions. As said before, the idea is to specialize them for not generating processing bottlenecks or undesirable delays. A specific agent named *Communicator* will be responsible for the identification and connection at RML, communicability, and organizational issues of the Smart Thing in the system. Another one type named *Interface* will be dedicated for interfacing sensors and actuators connected to a hardware controller. Besides, there will be *Standard* agents without any specific ability that are responsible for helping in the decision making the process. The Figure 4.2 depicts the architecture of a Thing, a Smart Thing, and how they interact with the RML.

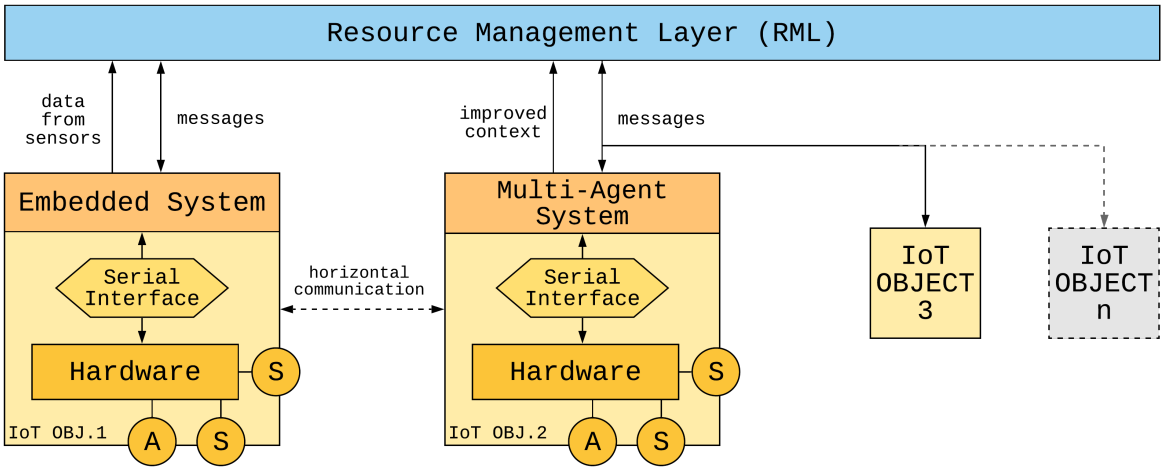

Figure 4.2: The architecture of a Thing and a Smart Thing interacting with the Resource Management Layer (RML).

A Smart Thing, as any IoT Object in the RMA, will configure itself in the system by informing its functionalities and purposes to the RML. Since it is autonomous, it could work independently from any environment or even if it is not connected to the RML. It is possible because the embedded MAS controls the architecture of the Smart Thing, and it can still be working even if it is not connected to any network. However, once connected, the IoT Objects can send data from sensors to the RML. In the case of Things, it sends raw data to be stored by the RML and consumed by clients that must use the raw data or treat them as well. In the case of Smart Things, it provides improved context data because the MAS can use the raw data perceived from its sensors to reason and, consequently, discover context. Besides, Things and Smart Things can interact with each other for exchanging information, but Smart Things can communicate to exchange beliefs, intentions, plans, and goals. Finally, clients can also send messages to the Things and Smart Things in order to execute some actions, for example. For this, they use the RML architecture, which abstracts the technological details from both clients and IoT

Objects. In the next section, the RML and the Application layer of the RMA will be explored.

4.3 The Resource Management Layer

The Resource Management Layer (RML) plays an essential role in the RMA because it separates conceptually and technologically both Application and Device layers dealing with all necessary mechanisms for managing IoT Objects connections and application services. The RML is the core solution of the whole architecture, and it keeps structures for the virtualization of IoT Objects — both Things and Smart Things — that will be consumed by clients using one of the exposed services provided in the architecture. As these services will access the RML concurrently, it needs to manage the resource availability for certain services for avoiding conflicts during the service execution, specially when contextual planning agents are accessing this layer since they need an assurance that the requested resources for executing a plan will be still available after the agent's planning mechanism is over. Therefore, the RML provides an interface for the Contextual Planning Layer (CPL) where agents can request for resources that are locked and allocated for a slice of time while they are using them.

The RML is also responsible for exposing services for data visualization in the architecture. These data visualization services include mobile applications and web pages that access the database of the virtualized components to retrieve information about environments and their IoT Objects. Hence, every IoT Object must be attached to a virtual environment to facilitate the management of resources since clients may access information considering environments, as CPL does, for example. Nevertheless, clients are not limited to retrieve information, and they can send action commands using a visualization service to interact with any public IoT Object registered in the RML. Besides, the RMA architecture is extensible for other solutions and services that want to exploit and make use of these functionalities explained before.

In order to clarify the approach, the RML architecture is divided into functionalities for explaining all those expected attributes. The first component of the RMA is the RMC, which manages virtualization and registration of physical environments and IoT Objects, providing information on their availability and allocating them to a client when necessary. The IoT Objects are dynamically registered in the RMC, at which point they inform all of their existing functionalities and characteristics. Once an IoT Object is registered, it

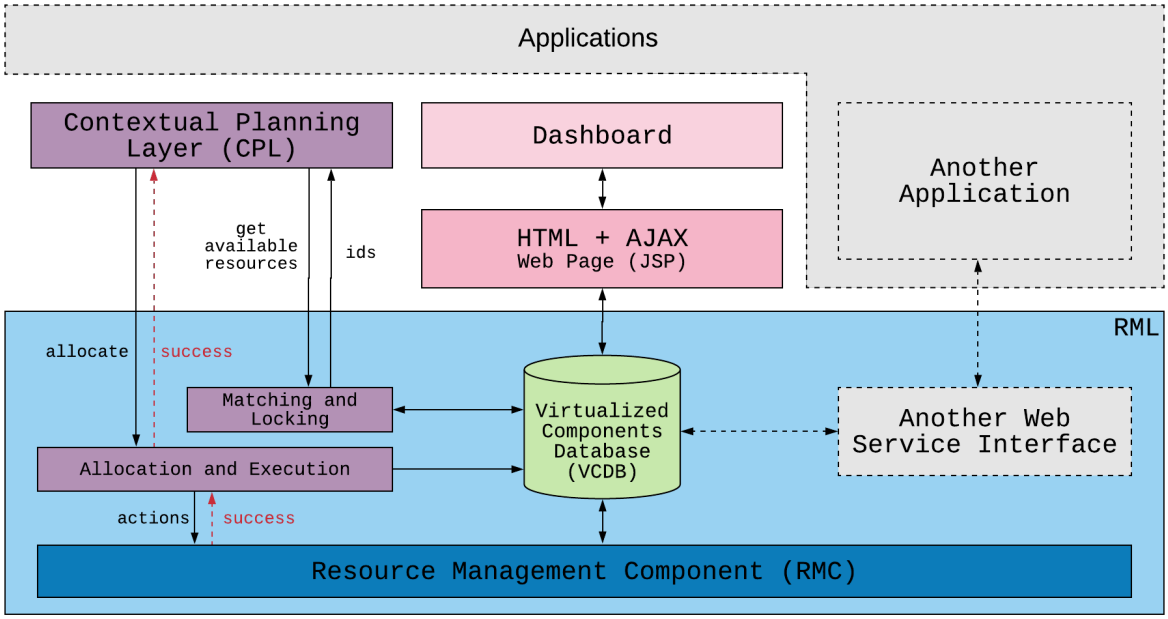

Figure 4.3: The components of the Resource Management Layer and its interfaces.

keeps updating the RMC with data gathered from its sensors. In case of disconnection, the IoT Object is unregistered from this layer. The RMC structure is capable of different mapping environments, each composed of IoT Objects and their capabilities. The RML functionalities are described below:

- **Environment Matching and Locking**: When the RMC receives a requisition for resource availability check, it compares the required resource sets with all available environments to determine which of them can be used. It then replies to the original request with the resource sets that can be provided. Also, the matching environments are locked until an allocation request is received, or a timeout occurs. Locking the IoT Objects guarantee the execution of a requisition from the CPL to use a specific resource set. This process is necessary to give the CPL enough time to evaluate plan feasibility, receiving confirmation from the agent on which plans will be executed (and which resource sets will be effectively necessary).

- **Allocation and Execution**: The Allocation and Execution module unlocks all the resources relative to the initial requisition that are no longer necessary, effectively allocating only the necessary resource sets. In the case there are no requisitions for allocating the locked environments after a specific time, a timeout process will occur (and all the locked environments will be released).

- **Virtualized Components Database (VCDB)**: The IoT Objects are continually sending data to this component, which always stores the last value for each mapped sensor (discarding the old ones). It also maps and manages every resource indicating if it is busy or not in order to avoid concurrency conflicts.

- **Applications and Services**: the RML allows several services to exist at the same time accessing the IoT Objects' resources registered at this layer. Despite the CPL service for contextual planning agents, mobile applications, and a web page for accessing the VCDB are interfaces for helping in data visualization. The RMC and the VCBD are prepared for receiving new interfaces or services if it is necessary or desired.

It is essential to observe that IoT Objects perform a dynamic configuration and update process when they connect in the RML for the first time. At this point, a connected IoT Object sends information about all its available functionalities (including all sensing and actuators' capabilities) and the current environmental conditions (perceived according to its existing sensors). Once connected, the IoT Object keeps updating the RMC with data gathered from sensors (once per device's cycle). Then, it checks if there are messages received from the RML and put them into a queue of actions to be executed at the end of the cycle. This process of our approach is explained in the next section.

4.4 An Architecture for the Internet of Smart Things

In the RMA, a device is an IoT Object that dynamically registers itself in the RMC by sending all its functionalities (available resources, data and action commands) and the environment where it is situated. Once connected, the IoT Object receives a confirmation, and it starts to send data gathered from its sensors directly to the RMC, which keeps the most recent value available to be consumed by the Application layer. Besides, the IoT Object deals with action commands coming from the RMC that need to be executed by its actuators. The RMC is the main component in the RMA, and it maintains information about IoT Objects and environments to be consumed as a service by other layers. It has a mechanism for registering IoT Objects in given environments and storing their information in the Virtualized Components Database (VCDB). The VCDB keeps updated the data coming from IoT Objects by considering only the new values that have changed since the last data message was received. Besides, the RML exposes services for providing

visualization layers for mobile applications or allowing new technologies to interact and access the VCDB. The detailed architecture is seen in Figure 4.4.

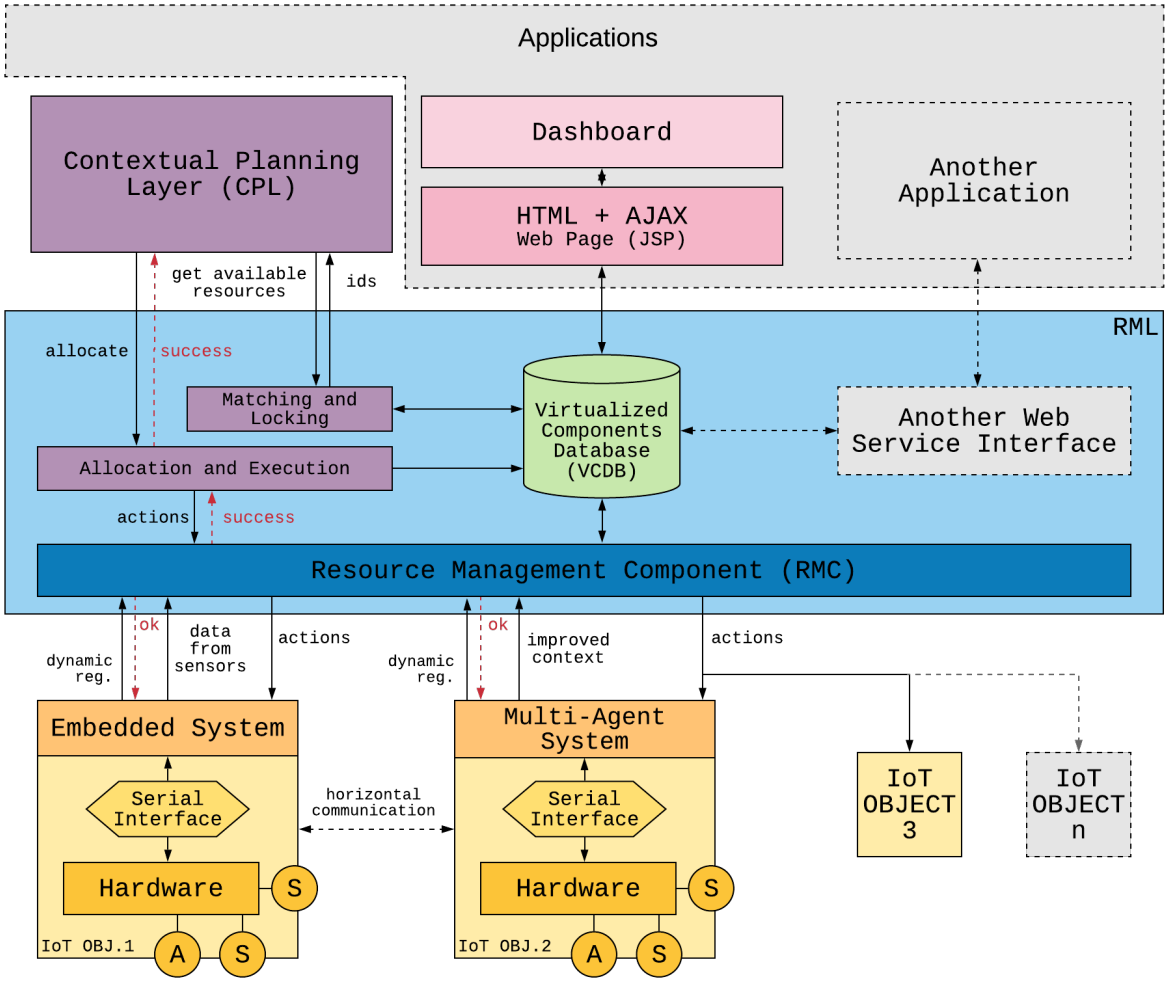

Figure 4.4: The Resource Management Architecture (RMA) overview.

The RMC also manages the actions that need to be executed by IoT Objects. Clients may perform actions in IoT Objects using some available application accordingly to their availability in the architecture since other clients can also be consuming resources from IoT Objects at the same time. When a resource is a sensor, it is normal that more than one client can access it at the same time. However, when it is an actuator, conflicts can arise if two or more clients try to access it at the same time. Furthermore, the RMC is responsible for forwarding the received action commands that need to be executed by a specific IoT Object since it keeps all of them mapped and registered in the RML. Applications exposed as services need to inform the action — such as turn something on or off — that they want to perform followed by the resource identification.

The RMC redirects the action directly to the specific IoT Object only in case it is available. One may notice that technical information about IoT Objects' hardware is transparent to clients, that need to know the available commands of the IoT Object it wants to interact. While one device is being used, the RMC blocks the access to this IoT Object for other clients for avoiding conflicts in the lower layer. The respective resource is unlocked after the action execution or after a timeout boundary. The RMC is a server-side solution running an IoT middleware instance where the devices must connect to work correctly. The middleware provides the connectivity and communicability necessary for IoT Objects to interact with the RMC layer, and it should guarantee the scalability of the system.

The Application layer is responsible for providing ways for clients to access the RML and consume data, as stated before. These accesses are materialized as solutions that allow both layers to communicate and exchange information. For facilitating the environments managing, the Application layer provides access to all environments using a web dashboard where users can create their public or private environments to share their own IoT Objects. This dashboard shows all the public environments allowing users to select and see their available information and subscribe to have access to a specific and personalized group of information. In this thesis, an environment is defined by a logical representation of a physical space where several IoT Objects can co-exist sharing information. Mobile applications have the same functionalities of the web dashboard. However, depending on the solution, personalized services can guarantee access to specific environments for facilitating their remote controlling.

Agents also play an essential role as clients of the RMA. Contextual Planning agents [21, 22] use resources as part of the reasoning process of evaluating the feasibility of plans to be performed in an AmI system. These agents need to know the availability and capacity of resources intended to be used, to define which plan is the best to be performed in the environment. Hence, the agents inform the resources needed for the execution of their goals, and the RMC responds, informing if there is any environment that matches the initial requisition. If so, this environment and resources are blocked until the agents decide which plan will be executed or a timeout boundary is reached. After that, the agent allocates the resources until the execution of the demanded plan's actions happens. At the end of this process, the initial resources blocked are free for the next client in the RMC.

Finally, the deployment of the architecture provides a decentralized AmI system,

where IoT Objects are heterogeneous, autonomous, capable of entering and leaving the system, and they have shareable resources that can be virtualized by a middle entity that mediates the data access for other clients. As these IoT Objects considers the heterogeneity in both hardware and software, the use of MAS on top of that creates a specific type of device named Smart Things. Then, in this thesis, the RMA is defined as the Internet of Smart Things (IoST), an IoT network of interconnected Smart Things and Things for supporting the development of AmI systems.

Chapter 5

The IoST Implementation

In this chapter, it is presented the implementation of all layers of RMA towards an IoST architecture for the deployment of open and dynamic AmI systems supported by embedded MAS as Smart Things. In order to accomplish this goal, it is essential to provide ad-hoc devices capable of reasoning independently from the proposed architecture, ways of communicating over the IoT network, and self-configurable mechanisms in the RMA.

It is discussed the technological issues of the overall architecture for the IoST, which is an interconnected network of Smart Things and devices based on the IoT for supporting the development of AmI systems. The architecture considers an open and dynamic environment where devices using the agent approach (Smart Things) and other devices can enter or leave anytime. This approach leads us to a decentralized architecture and collective reasoning since every embedded MAS is considered an autonomous and independent thing capable of communicating and negotiating — or even acting as a group pursuing common goals — with others devices. By independent, it means that Smart Things can keep running and reasoning even if communication and interaction technologies stop. It is important to remark that it is only possible in cases where the reasoning process does not depend on information coming from other Smart Things. For example, if a Smart Thing is responsible for identifying if a room is cold or warm, and it keeps the temperature straight by controlling air-conditioners, the Smart Thing will still perform this operation even if there is none communication with other layers.

5.1 The Structure of Things and Smart Things

In a common architectural approach for creating IoT and AmI systems is expected that on a bottom layer several sensors and actuators are controlled by different microcontrollers,

while on a middle layer, generic services for accessing sensors and actuators are implemented and provided for applications running on a top layer. Thus, it is essential to adopt a structure for creating IoT Objects able of controlling microcontrollers with connected sensors and actuators, that uses embedded systems with enough storage and processing power for dealing with all functionalities of Things and Smart Things, and communication mechanisms to interact with other components. As the IoST is a layered architecture where IoT Objects has to connect to a middle layer — the RML — for exposing their resources, it is necessary to provide a structure that complies with these needs.

Hence, Things and Smart Things of IoST are built over a heterogeneous structure, which can control different microcontrollers in the hardware layer. This structure could employ microcontrollers of different types, each of them controlling sensors and actuators in the same IoT Object. We consider being heterogeneous a structure that can employ different types of microcontrollers in the same or different IoT Objects, and the type chosen do not interfere in the embedded system [91]. For this, a hardware interface is responsible for exchanging information between the hardware and the embedded system using a protocol that isolates both technologies. The embedded system has similar functionalities in both Things and Smart Things, it is responsible for capturing the sensors' data, it sends commands to be executed by actuators, and it deals with the communication and connectivity in the RML. Moreover, Smart Things uses a MAS, as an embedded system, responsible for controlling all hardware components providing autonomous and cognitive reasoning. Besides, the MAS of a Smart Thing has specialized agents for performing hardware interfacing and communication.

Thus, the structure of Things and Smart Things is modular, and it includes a module for the embedded system and the MAS; one for the hardware components with all microcontrollers, actuators, and sensors and; and the serial interface responsible for the data transferring between software and hardware modules [63]. The modules are created separately, and they communicate by using the hardware interface with a defined protocol. So, the embedded system and the MAS are independent of the hardware, and they both can be modified or changed without interfering in each other. The microcontroller is programmed to send all the sensors' values using the hardware interface to the embedded system, and all possible actions that can be executed are also planned. The embedded system, some functions or agents, in case of MAS, are designed to interact with the microcontrollers sending commands to the hardware module for requesting data or execute actions.

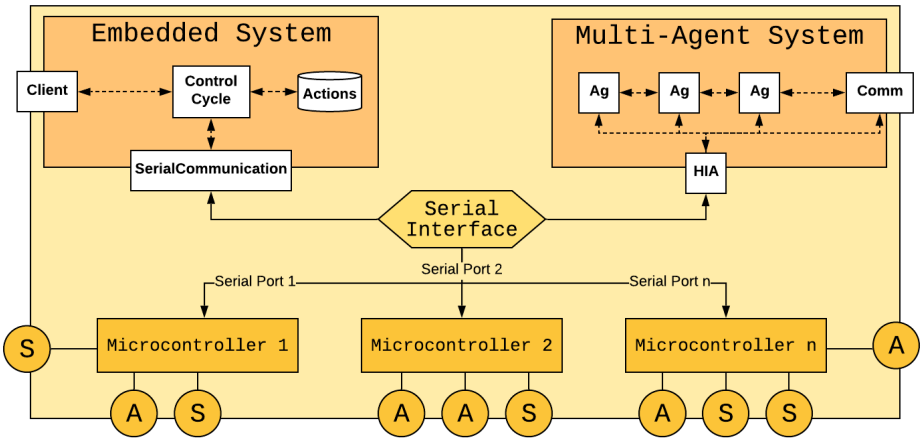

Figure 5.1: The layered architecture representing both Things and Smart Things interfacing hardware. The serial interface is common to both IoT Objects, and Things use an embedded system while Smart Things use MAS.

The IoT Objects needs to execute Java-based application in the embedded system, in case of Things, or any MAS framework based on Java, such as Jason [11] or Jade [7], since the middleware employed in the RML and the hardware interface developed for IoT Objects (Section 5.1.1) are Java-based libraries. In Smart Things, it is used the Jason framework and hardware interface agents (Section 5.1.3.1), for controlling hardware's resources. In the context of software agents, the use Jason and hardware interfacing agents provides a platform which supports the developer to program AmI Systems solutions without concerns about integration issues between hardware and software because of the independence between the layers. All perceptions are directly processed without the intervention of the designer. Besides, the capability of using different microcontrollers in the same project controlled by a MAS is the main contribution of using this approach [89].

IoT Objects can communicate with each other and connect to the RML to expose their resources. For this, the ContextNet middleware works as the communication channel between all the IoT Objects. The embedded system of a Thing employs a client instance of the middleware to provide connectivity and communicability with the RML. It also implements a cycle to synchronize the behavior of Things and to avoid undesired conflicts in mounting, sending, and receiving messages. Besides, the RML is built over a core instance of ContextNet where is kept the information of IoT Objects' resources. In Smart Things, another significant improvement is that a customized and specialized agent is responsible for the communicability and connectivity in the RML. This type of agent has a client instance of ContextNet inside its reasoning cycle, allowing to exchange beliefs,

intentions, goals, and even plans at runtime. In the next sections, all the technical details of implementing Things and Smart Things will be explained and described.

5.1.1 Hardware Interface

Both Things and Smart Things need to access hardware — sensors, and actuators — to retrieve information or to perform some action in a given environment. Hardware-Software Interfaces are technologies that deal with the communication and data transferring from hardware devices to a software solution. As a Thing or a Smart Thing are composed of hardware components interfaced by microcontrollers that send data to an embedded system hosted in another tiny computer, it is essential to provide a bridge using these kinds of interfaces for guaranteeing a proper communication between hardware devices. Sometimes, middleware is also used for these purposes.

However, these components mentioned above are connected through serial wires limiting the range of solutions that could be employed. For instance, Bluetooth and Wi-fi connections are excluded from them. The wire connection is the most appropriated interconnection in this case, since the components are nearby each other. Some of the available solutions for this purpose do not deal with the duality hardware-software dedicating efforts for only one side (hardware or high-level languages) such as RxTx Library and JavaComm. There is middleware with fancy functions that provide communication and data transferring along with different layers of abstraction in hardware development such as [96]. However, they demand high computational processing for these kinds of platform such as Things, and it could add serious delay for MAS considering a Smart Thing.

Hence, we developed a double-sided and generic serial interface for data exchanging and communication between microcontrollers and high-level languages that implements a protocol with error detection and discard of messages ensuring that the receiver will not accept messages with data loss. This protocol aims to guarantee that messages with the loss of information arrive at the embedded system, avoiding unexpected crashing, especially when talking about MAS. As the protocol considers only the message content that is traveling between both sides, the technology employed in the low-level hardware is not related to the high-level language employed in the embedded system. These characteristic endorses the functional attribute of Heterogeneity, and it helps in achieving the non-functional attributes of Reliability, in exchanging messages from hardware to software, and Adaptability as mentioned at Chapter 4.

Before sending a message to the software, the microcontroller mounts the message, calculating the content size that has to be sent. It adds a preamble for checking the correctness of the message on the other side. Then, when the hardware side library sends a message, the software side library receives and verifies the correctness of the content. If there is no data loss during the transmission, the message is delivered. When the software side library needs to send a message to the other side, it is necessary to identify the port where the target device is connected.

When using this interface, the message follows the structure with three fields: **Preamble**, composed by 2 bytes of a fixed value to identify a new message; **Size** that has 1 byte used to inform the size of the message sent and; the **Message** (up to 256 characters) to be sent [63]. The two firsts fields, **preamble**, and **size** are both used to identify errors that can occur because of collisions or information loss during the message transmission using serial ports. At the end of the transmission, the receptor verifies if the preamble received is the expected one. If it is, the receptor takes the field **message**, calculates its size, and compares this result with the field **size**, which composes the message. If the calculated value from the message is equal to the field **size**, there is no error, and the receptor can use the message. Otherwise, if the received preamble is not the expected, the protocol discards the message.

The double-sided interface is composed of a library for microcontrollers and another library to be used at high-level languages. For now, it is provided libraries respectively for ATMEGA and PIC controllers and for Java, which are the main technologies used in Things and Smart Things. To perform serial communication between ATMEGA or PIC microcontrollers and the Java programming language, the low-level library is a C-based library (one for each microcontroller). The library was developed to work in controllers with at least 256 bytes of RAM because the size of the message is up to 256 bytes. The methods implemented for the low-level side are:

- **sendMsg(String msg)**: This method sends a message to the software layer using serial communication;
- **availableMsg()**: This method checks if exists messages coming from the software layer. The method returns a boolean value informing whether there is a message available or not;
- **getMsg()**: If there is a message available, this method is used to get the request information sent by the software to perform some action using actuators or to gather

perceptions.

The Java-side library offers four methods described as follows:

- **sendCommand(String port, String msg)**: This method sends a message to the hardware that is connected to the serial port informed and returns a boolean value informing if the transmission occurred without a problem;
- **requestData(String port, String msg)**: This method also receives as a parameter a message and the port to be used to send that message, but, unlike the previous one, it expects the hardware to send back a response message that needs to be programmed in the hardware side. The method returns a boolean value informing whether the hardware sends a response or not;
- **listenHardware(String port)**: This method checks if exist any message sent by the hardware in the informed port;
- **getData()**: This method is used to get the information sent by the hardware when one of the previous methods inform that there is a message available.

Based on the libraries implementation, it is possible to employ the interface using operation modes. Three operation modes can be programmed to determine the behavior of the hardware concerning the software. The microcontroller works in a master-slave way, receiving commands from the software side or sending asynchronous messages. The operation modes for determining how the hardware will operate are:

- **Request**: it is a synchronized process of message exchanging where the software requests all the sensors' data from that specific microcontroller and the hardware answer it with a formatted string containing the information. This mode guarantees a response for every request made.
- **Listening**: it is an asynchronous process of exchanging messages where the microcontroller is programmed to send the sensors' data continuously, and the software tries to reach this information when needed. In this case, there are no guarantees that the message will arrive when it is needed.
- **Send**: it is responsible for receiving commands that will activate actuators. The microcontroller is prepared to get the command and call a function specified by the system designer.

Despite the use of the interface in for creating Things and Smart Things, it was used in several other domains such as prototypes for educational purposes [37, 42] and employing MAS as well [86, 14, 69].

5.1.2 The Embedded System of a Thing

Things are IoT Objects composed of a tiny mobile board with Bluetooth and WiFi connections with enough processing power for running an embedded system (e.g., Raspberry Pi) connected to one or more microcontrollers using serial communication for accessing sensors and actuators. The microcontroller is programmed in a loop for verifying if requests are coming for gathering data or executing actions. If it is a request for gathering data, it accesses all the sensors and mounts a string to be sent by serial communication to the embedded system. Otherwise, the action received is verified, and executed if exists an equivalent one expected, in the respective actuator. Actions, in this section, are any commands that need to be executed by Things.

As stated before, the embedded system of a Thing deals with the connectivity and communicability with the RML since it employs a client instance of ContextNet. It gathers data from the sensors using the Hardware Interface and how to send the data acquired to the RML. It also receives actions from clients or other IoT Objects to be executed by actuators. These actions are redirected to the microcontroller using the Hardware Interface library. All these steps happen during the execution of the cycle of the Thing (Algorithm 1), which controls its behavior. First, there is a configuration file where the developer of IoT Objects informs all functionalities that a Thing possesses at design time. These functionalities are every resource, available actions that Users can interact with, and where physically the resources are connected to since the Thing can employ more than one microcontroller.

IoT Objects are capable of self-configuring in the RML, and the information contained in the configuration file plays an essential role in this process. This file is processed by the embedded system, which logically mounts the device in memory, and it sends this information to the RML, which stores them in a specific database of virtualized components, the VCDB. The RML answers it with an acknowledgment message. The IoT Object cannot send any data to RML while it is not registered. It is important to remark that this process is similar to Smart Things.

After the device is registered at the RML, the cycle synchronizes the reception of data coming from sensors to be sent to the RML and the actions that need to be sent

Algoritmo 1 Embedded System's Processing Cycle of Things.

```
1:  procedure CYCLE(configurationFile)
2:      mountDevice(configurationFile)
3:      intervalTime ← getIntervalTime()
4:      loop
5:          if isRegistered() then
6:              getheredData ← dataFromSensors()
7:              sendToRML(gatheredData)
8:              wait(intervalTime)
9:              actions[] ← getReceivedActions()
10:             executeNextActions(actions[])
11:         else
12:             registerDevice()
```

and executed in the microcontroller because both cannot execute at the same time for avoiding undesired conflicts. Hence, the actions received from the RML are put in a queue of actions to be executed one step after the data from sensors are collected through serial ports. The cycle also counts with an interval time between sending the gathered data to RML and receiving actions that must be set by the IoT Developer at design time. It does not have any practical influence in the process, but it works as a variable to control the pace of the Thing, if necessary. The whole interaction between Device and Cloud can be seen in Figure 5.2.

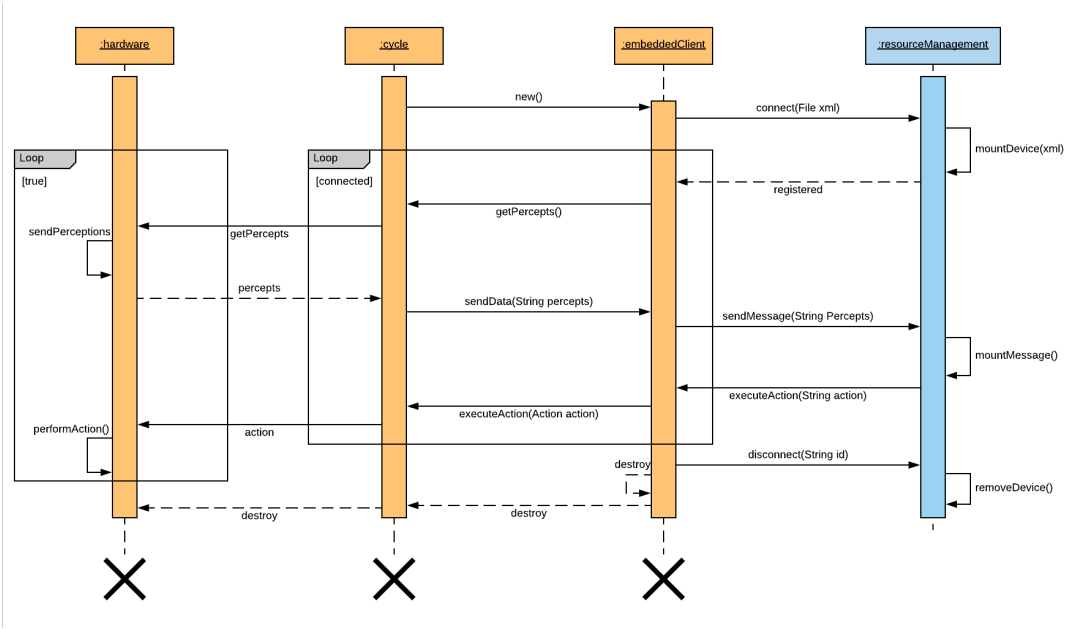

Figure 5.2: The activity diagram representing the message exchanging between Things and RML.

As stated before, Things are capable of performing a self-configuration and registration at RML when it starts running, and there is a server available. For this, the Thing

keeps an XML file describing all the available resources, commands, and also the server configuration information. Once the file is processed, the embedded system's cycle performs everything automatically. Hence, the file must properly be filled in order to deliver the best performance of the approach. The following information must be provided:

(i) **Server**: the gateway of the server — where the server instance of ContextNet is installed — and the port must be provided to connect at RML. Besides, the time interval of sending data to the server must be set.

(ii) **Environment**: for instance, it is necessary to inform in which environment the IoT Object is inserted into since there is no automatic environment discovering a process, or indoor/outdoor positioning system available. For this, the identification number of the environment must be declared in the configuration file. The environment's identification is generated when Users register an environment in the Management Dashboard system.

(iii) **Resources**: all the resources available must be mapped in the configuration file as well. All the resources have the serial port where it is connected, its name, and a non-mandatory description. If the resource is an actuator, it also has the available execution commands (actions).

Finally, the embedded system uses a model composed by the *Cycle* class, which instantiates an instance of ContextNet client (represented by the *EmbeddedClient* class) capable of exchanging message with the RML. Besides, the *EmbeddedClient* is responsible for interfacing hardware components, and it deals with a queue of actions received from RML that has to be executed. The *SerialCommunication* and *Action* are the classes responsible for the Hardware Interface implementation and the unit of action respectively.

Both Things and Smart Things shares some of the technological details for registering in the RML and interfacing hardware. The configuration file process, the hardware interface, and the use of an instance of ContextNet clients are used together with agents. Smart Things use embedded MAS, where there are specialized agents for each one of these functions. Besides, Smart Things do not use a cycle, since every agent possesses an associated reasoning cycle. In the next section, the implementation of MAS for Smart Things will be described.

5.1.3 The Embedded MAS of a Smart Thing

The main difference between a Thing and a Smart Thing is that the former one uses an embedded system for controlling its functionalities, and to provide ways of controlling the hardware for accessing data and actuators while the later one it is enhanced with an embedded MAS for the same purpose adding autonomy and intelligence for such kind of IoT Object. The advantages of employing MAS in Smart Things were discussed in past chapters. This section intends to focus on the implementation of the technologies for embedding MAS.

There are plenty of agent-oriented languages and frameworks for programming MAS [10] but they do not provide internal hardware interfaces in their architectures leading to situations where external hardware interfaces must be used along with MAS. As these interfaces are not specific for this purpose, they can introduce delays and errors, which are undesired issues for any system, including MAS. Some of the agent-oriented languages are specific to programming robotics agents [53, 27, 35, 126] and they mainly focus on creating and interfacing robots, which demand high-processing and robust solutions in many different fields of computing. However, mainly, they consider it as one robotic agent instead of a MAS capable of communicating with other devices in an AmI system and, regarding communication, they need to make use of robotics middleware that is not specific for IoT purposes.

So, to provide Smart Things for IoT and AmI systems, it is necessary to employ an agent-oriented language capable of interfacing hardware information directly to the internal structures of the language in order to gain performance in processing data as perceptions and beliefs, once it is being used the BDI. Besides, it needs mechanisms to connect to an IoT network, specifically to the RML, to expose their resources to be accessed by clients, and ways of communicating with other Smart Things for transferring beliefs, goals, intentions and plans as part of the learning process inherent to Smart Things. There is no such approach that considers both hardware interfacing and communication in the same language.

Therefore, in order to provide those attributes for MAS, two new types of customized agents were developed using the Jason framework. The Hardware Interfacing Agents are capable of interfacing sensors and actuators using the double-sided hardware interface described in section 5.1.1 in the agent's reasoning cycle. All information that comes from the interface is processed as perceptions in the agent's internal belief base, and every action that is needed to be executed is sent by the agent using their plans. This is a

practical approach for both the programmer and the agent architecture since they do not need to be aware of the technological details of interfacing hardware and the system gains in performance and reliability because it is directly processed [57, 86].

The other type of customized agent is responsible for the communication with other IoT Objects and the connectivity of the embedded MAS — consequently the Smart Thing — in the IoST. Similarly, the reasoning cycle of the agent is extended with a client instance of ContextNet, becoming possible to send and receive messages from the IoT network. In the same way, it is also possible to connect and send information to the RML. This extended ability of communication allows a closed MAS — which does not allow interaction with other agents except those from its system — to interact with other closed MAS. It allows that two distinct and embedded system can exchange information without being bounded technologically. Those both agents extensions endorse the functional attributes of Autonomy, Communicability, and Connectivity described in Section 4. In the next sections, we describe these customized agents in details.

5.1.3.1 Hardware Interfacing Agents

Differently from Things that uses an embedded system where the hardware interfacing occurs inside a defined cycle, Smart Things uses a MAS composed of several agents with their reasoning cycles that interacts for controlling all functionalities of the IoT Object. The MAS is developed using Jason framework, which does not have an interface to capture perceptions directly from the real world in its distribution. For this purpose, it was necessary to create an extension of Jason's agents capable of controlling resources such as sensors and actuators connected to microcontrollers [89, 90].

This extension can capture perceptions and send them to a specialized type of agent, which process all the information directly as beliefs in its belief base. In the same way, the agents are able of executing actions using actuators without worrying about what kind of hardware technology is being employed since the reasoning cycle of this — from now on named Hardware Interfacing Agent — uses the Hardware Interface (Section 5.1.1) instead of using the simulated environment provided by Jason in the capturing perceptions step. It provides serial communication between microcontrollers and Jason using a basic protocol to guarantee the correct information exchange between the transmitter and the receiver.

In order to create the Hardware Interfacing Agents, a customized and extended architecture of the reasoning cycle of a traditional BDI agent of Jason is provided, and it can be seen in Figure 5.3. There are two points where the cycle is modified from the original one to allow the information to be processed as beliefs and to send action messages to activate any actuator. At the beginning of the reasoning cycle, it is performed the act of perceiving the environment, at this moment, the Hardware Interface captures the information coming from sensors to be processed in Belief Update Function (BUF). However, before this, the information can be filtered by the agent. The filtering process works in cases where there are many beliefs that can trigger undesirable events on that occasion. Therefore, the Hardware Interface is also employed at the end of the reasoning cycle in a step responsible for actions that the agent has to perform. It allows agents to send actions to a specific microcontroller using the Hardware Interface directly from its plans.

A Hardware Interface Agent has the abilities, at runtime, of selecting the microcontroller which it desires to control. In fact, a MAS using this type of agent can have many microcontrollers, and the agent chooses which one it wants to control by pointing to the serial port where it is connected; it decides whether or not to block the beliefs coming from sensors releasing processing time for other tasks, for example; it filters undesirable perceptions using perceptions filters created at design time; it limits the time interval of perceiving the environment. Besides, a MAS can be composed of traditional Jason's agents and Hardware Interfacing Agents working simultaneously. The traditional agents can perform plans and actions only in software level and communicate with other agents in the system (including hardware interfacing agents).

The Hardware Interfacing Agent is a traditional agent with additional features, such as the ability to communicate with the physical environment, perceive it, act upon it, and filter information perceived from sensors connected to microcontrollers. However, these agents only communicate with agents hosted in the same MAS. This implies that if a MAS is embedded into a Smart Thing, which is programmed with only these two types of agents, the communication will be limited, internally, to this device. In order to facilitate the programming of these agents capable of interacting with hardware, the Hardware Interfacing Agent has five internal actions that can be used:

1. **.port(Port)**, where the agent chooses which device to control selecting the serial port where the device is connected (e.g. .port(com8));
2. **.percepts(open or block)**, where it is defined if the agent blocks or releases the

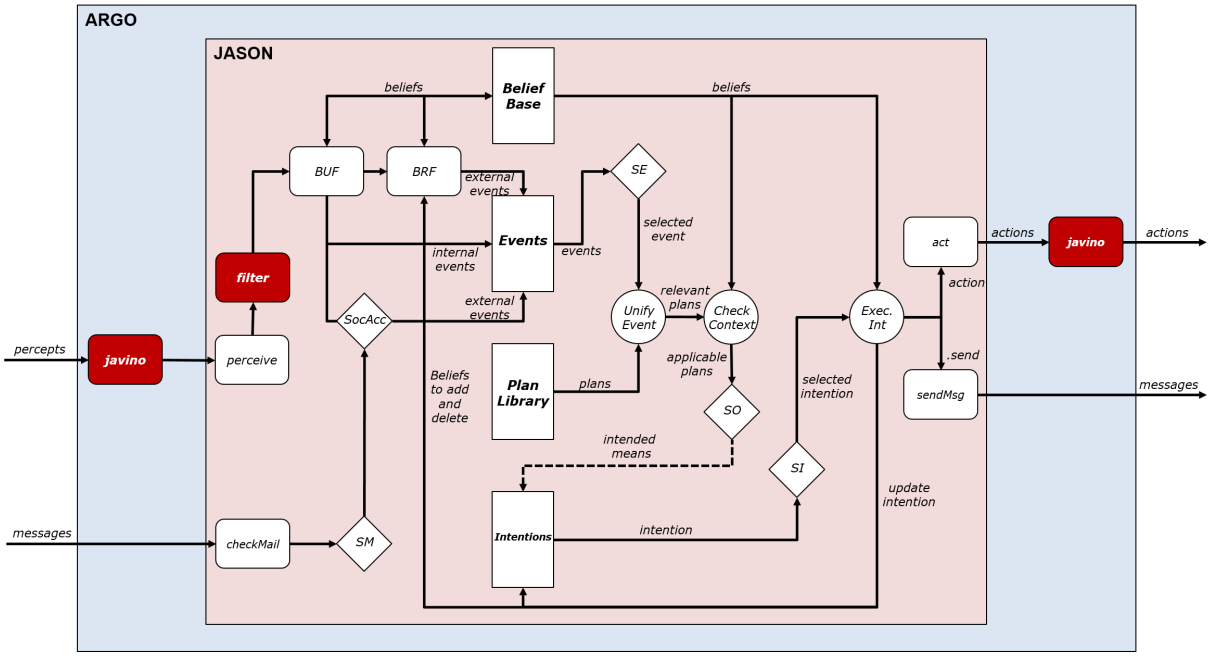

Figure 5.3: The reasoning cycle of a Hardware Interfacing Agent [89].

flow of perceptions from the controller;

3. **.limit(milliseconds)**, which defines for how long the environment should be perceived;

4. **.act(message)**, which sends a message through the serial port to execute an action using an actuator and;

5. **.filter(XML)**, which selects the XML file responsible for filtering perceptions.

Despite the use of The Hardware Interfacing Agents in Smart Things, this approach has been used and tested in the prototyping of smart homes, autonomous vehicles, and in bio-inspired solutions based on MAS. However, there is a lack of communicability if an embedded MAS is employed only with Hardware Interfacing Agents. In the next section, it will be described as another customized architecture of agents that will be responsible for connectivity and communicability in the RML.

5.1.3.2 Communicability and Connectivity for Smart Things

Once there are Smart Things with the ability of interfacing hardware, now it is necessary to provide communication abilities between different devices embedded with MAS

for truly creating Smart Things. In some cases, the use of MAS brings some advantages compared to IoT Objects that only work as data repeaters sending information from their sensors to a server for discovering context about a situation and need stimulus from other components to act upon the environment. Agents are pro-active, autonomous, and are capable of reasoning about information from the environment that they are situated. In this section, it is described the development of a new kind of agent named *Communicator*, which is able of communicating with IoT Objects in the IoST and interacting with the RML using *ContextNet* as a communication channel.

In order to allow the programming of agents that are able of communicating through IoT, a special kind of agent named *Communicator* is developed. This agent is responsible for exchanging messages with agents hosted in a different MAS or Smart Things. It is important to remark that traditional agents can only communicate with agents from their own MAS. Since the communication is done using ContexNet instances — both in the IoT objects and RML — the *Communicator* agent must have a communication mechanism for sending and receiving messages through the IoT. Thus, the reasoning cycle of the *Communicator* agent was extended with a client instance of *ContextNet* middleware embedded in its architecture (Figure 5.4).

The first modification happened at the beginning of the reasoning cycle where now agents are capable of receiving messages from others IoT Objects using *ContextNet* Client Library and messages coming from agents of its own MAS using the native *checkMail* method. All messages received are processed to generate events and update the agent's *Belief Base*. The next modification was inserted at the end of the reasoning cycle after the *sendMsg* step. At this moment, the agent can send a message to agents hosted in its MAS or to another IoT Object in RML using *ContextNet* Client Lib. A message can be sent to another **Communicator** agent or any IoT Object able of understanding the message format since agents in Jason uses the KQML agent communication language. For this, a mechanism for translating messages from agents to IoT Objects are necessary to guarantee the interoperability between platforms. In the case of Smart Things, it is not necessary since both use Jason, and they communicate using the same language.

Every *Communicator* agent must have a unique identification number provided by *ContextNet*. This identification number guarantees that the Smart Thing will be uniquely identified in the RML and the IoST. Hence, it leads to a MAS configuration where is allowed just one *Communicator* agent per Smart Thing that will be responsible for all communication with the IoST components. In order to send any message, the agent uses

a new internal action named **sendOut** developed based on the original internal action **send** from Jason. The major difference between them is that **sendOut** sends a message to a **Communicator** agent in another MAS embedded in a Smart Thing or other IoT Object with a translator mechanism. It works by sending a message to an addressee — other IoT Object — using an illocutionary force. The available illocutionary forces to be used along with *Communicator* agents in MAS are:

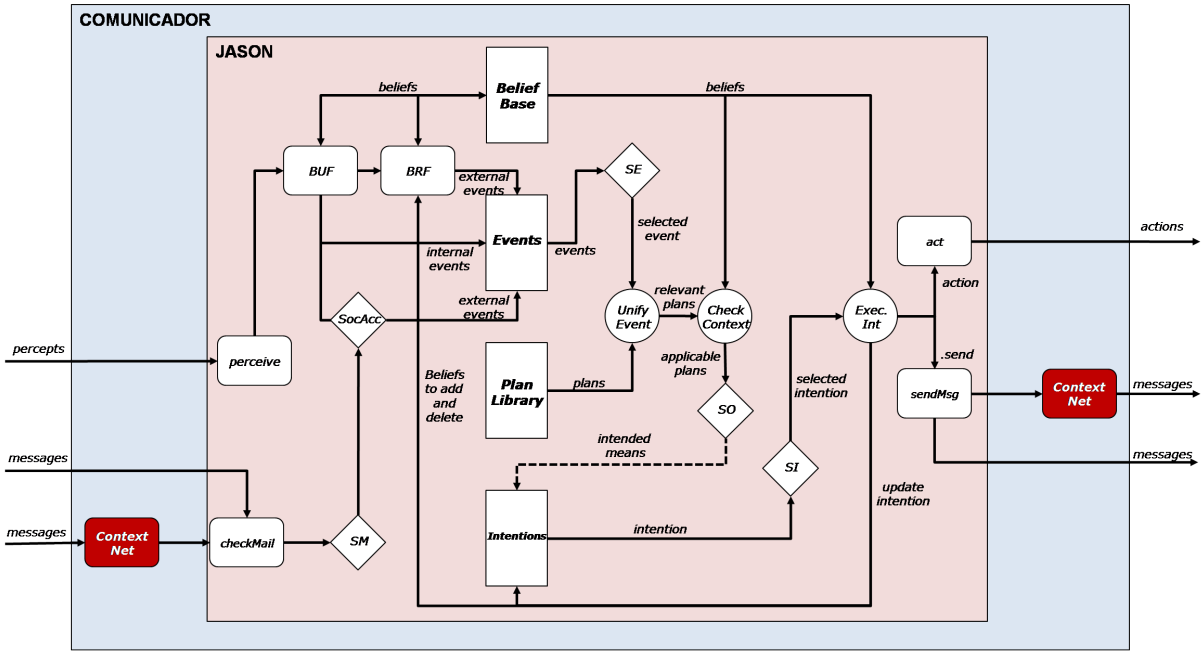

Figure 5.4: The Reasoning Cycle of a Communicator agent.

- **achieve**: sends a goal to be accomplished by the addressee. The content of the message sent will be inserted in the base of intentions of the addressee agent.
- **unachieve**: drops a goal if it has not been reached yet. The content of the message will be removed from the base of intentions of the addressee.
- **tell**: sends a belief of the sender to the addressee, which beliefs to be true. The content of the message must be a literal, which represents a belief and will be inserted into the belief base of the addressee.
- **untell**: the sender agent informs the addressee agent that the belief is no longer to be believed. The content of the message is removed from the belief base of the addressee.
- **askHow**: it asks the addressee if there is any implementation for a requested plan. The addressee sends all the implementation available for the requested plan.

- **tellHow**: sends the implementation of a plan to the addressee. The addressee does not need to ask for the implementation first.

In order to integrate *ContextNet* into Jason architecture, some modifications were performed. First of all, the *Communicator* class for creating an agent with the ability to communicate was added as an agent's customized architecture. This class has an attribute *commBridge*, which is responsible for all the functions of sending and receiving messages from *ContextNet*. This class also has a function for adding the received message from *ContextNet* to the agent's mailbox. The *commBridge* implements a process for mounting and verifying a message to guarantee no losses of data in the communication based on a protocol similar to the one from the Hardware Interface. A message is composed of the following fields: a preamble to identify the origin of the message with a length of 4 bytes; fields to identify the sender and the receiver of the message with 32 bytes each; a field to identify the illocutionary force with 32 bytes; and a field with the message content with 256 bytes.

When the sender mounts the message, it calculates the size of all fields to identify the beginning and the end of each field. After that, the preamble is added at the beginning of the message to verify the origin of the message. The message is mounted adding all the fields in a single string message that is sent by *ContextNet*. When the receiver receives the message, the preamble is verified to guarantee the origin of the message. Then, all the fields' size is verified to guarantee no losses in the communication process. If everything is right, the message is mounted and processed as Jason's message. Otherwise, the message is discarded. Figure 5.5 illustrates the steps of processing of a message by Jason.

The native *TransitionSystem* class of Jason was modified in the reasoning cycle function for allowing to check if exist messages to be read coming from *ContextNet*. The existing messages are added to the mailbox of the agent to be processed as beliefs or intentions depending on the illocutionary force related to the message, as explained before. After that, the agent can send a message using an internal action named *.sendOut*, which uses the *.commBridge* to send a message using the *ContextNet*. Another internal action named *.setMyCommId* is responsible for setting the identification string used by *ContextNet* to identify uniquely a Smart Thing in the IoST. It is important to remark that all modifications proposed do not interfere with the original Jason distribution nor in Hardware Interfacing Agents.

As Smart Things are also components of IoST, they need to connect to the RML in order to expose their resources. For this, there is necessary to provide three new internal

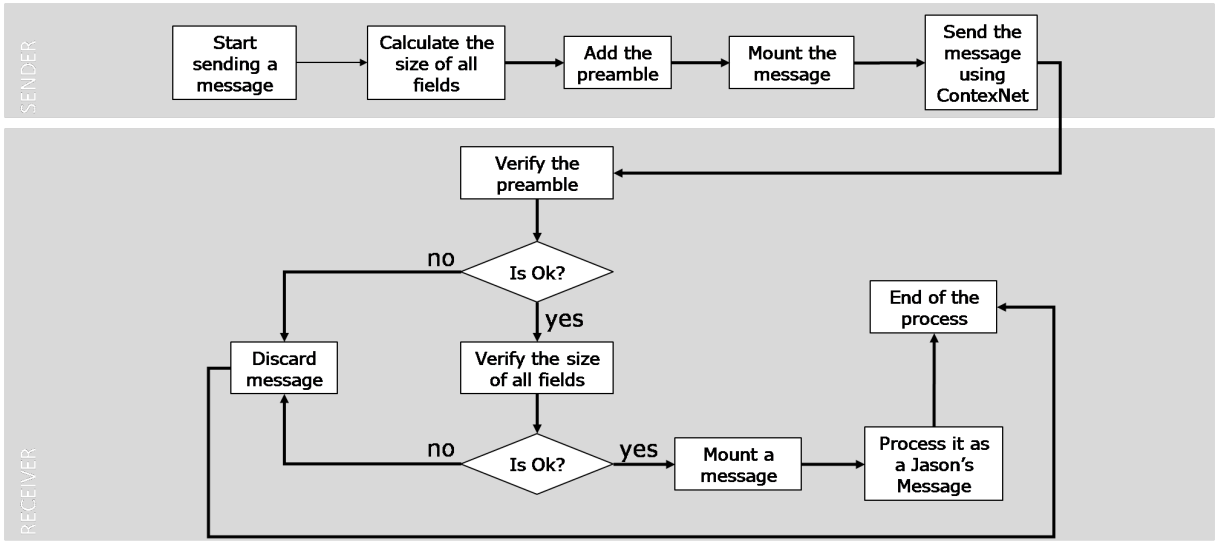

Figure 5.5: The process of a ContextNet message in Jason.

actions to allow Smart Thing to connect and disconnect to the RML, and to send the values from its resources to keep them updated in the RML. The internal action *.connect* is responsible for connecting the Smart Thing in the given gateway where the RML is hosted using the configuration file, the same used in Things, and the internal action *.disconnect* removes all the related data of the Smart Thing from the RML. Finally, the internal action *.sendToCloud* sends the data-informed in the configuration file to be stored and updated in the RML.

It is important to remark that our approach allows designing Smart Things able to interact with other IoT Objects (including another Smart Thing) using **Communicator** agents, which can connect to the RML and communicate to others IoT Objects using the *ContextNet* in the IoST. Besides, there is another type of agent responsible for controlling sensors and actuators, presented in the past Section, that can be used along with **Communicator** ones. Therefore, for programming the embedded MAS for Smart Things, it is possible to employ four types of agents:

- **Standard**: the standard agent can communicate with other agents of its MAS, but it is not possible to communicate with agents from a different MAS, and it is not able to control any hardware. It is the basic unit of a MAS. In the IoST architecture it is equivalent to *Traditional* agents.

- **Hardware Interfacing Agent**: it is a customized architecture of agents capable of controlling microcontrollers independently of its type and the domain applied in

the solution. These agents have all the abilities of a standard agent but are not able to communicate with agents from a different MAS.

- **Communicator**: this agent can communicate with agents from a different MAS or any IoT Object using *ContextNet*. It has the same abilities as a standard agent, but it is not able of controlling hardware devices.

The *ContextNet* is a scalable middleware, which guarantees a significant number of devices transmitting data at the same time. In this approach, we propose its use to exploit some advantages of using a robust middleware for IoT applications along with a well-known framework for agents solutions. Then, creating a customized architecture for agents **Communicator** with a *ContextNet* instance embedded into its implementation allows Smart Things to connect, interact to the IoST, and to communicate with other IoT Objects, or even systems if they use the same language. If a system needs to communicate with a Smart Thing, it should send messages in KQML format and translate the received KQML messages. Several works deal with the interoperability between MAS [114, 44, 107]. Then, it is out of the scope of this thesis to propose mechanisms for processing messages based on agent communication languages. In the next Section, the main components of the RML and the IoST Clients of the Application Layer will be described.

5.2 The Resource Management Layer (RML)

In the IoST, it is possible to use mobile and fixed Things and Smart Things that can enter or leave the system or that are fixed in the infrastructure of the environment and cannot be moved without changing physical parts, such as electrical facilities. Physically, these IoT Objects are composed of: the platform for the embedded system, which could be any tiny computer, such as Raspberry Pi, or any computational platform able of hosting an operational system; sensors and actuators; and microcontrollers. In this approach, we assert that Smart Things can employ embedded MAS for managing sensors and actuators in the AmI systems using Jason framework enhanced with agents responsible for interfacing with hardware components and agents responsible for communication in the network using an instance of *ContextNet*.

Basically, every IoT Object should connect to a server computer running a solution developed over the *ContextNet* middleware for IoT in order to communicate with other

IoT Objects and to make available its resources to be accessed by IoST Clients that can be Clients, Services, or any other application, such as a Management Dashboard or a Web System for data visualization. In this section, it is presented the implementation of the Resource Management Layer (RML) considering technological components and how these layers interact with the Device and Application layer. Figure 5.6 depicts an overview of the IoST focusing in technological components. For both Device and RML, it is employed the ContextNet middleware [48], which is an IoT middleware for context reasoning and data sharing in a large scale environment. It was chosen because it is a context-providing service for stationary and mobile networks, which already addresses data communication issues such as fault tolerance, load balancing, and node disconnect support (handover). It also uses the OMG DDS [93] protocol for handling messages between clients.

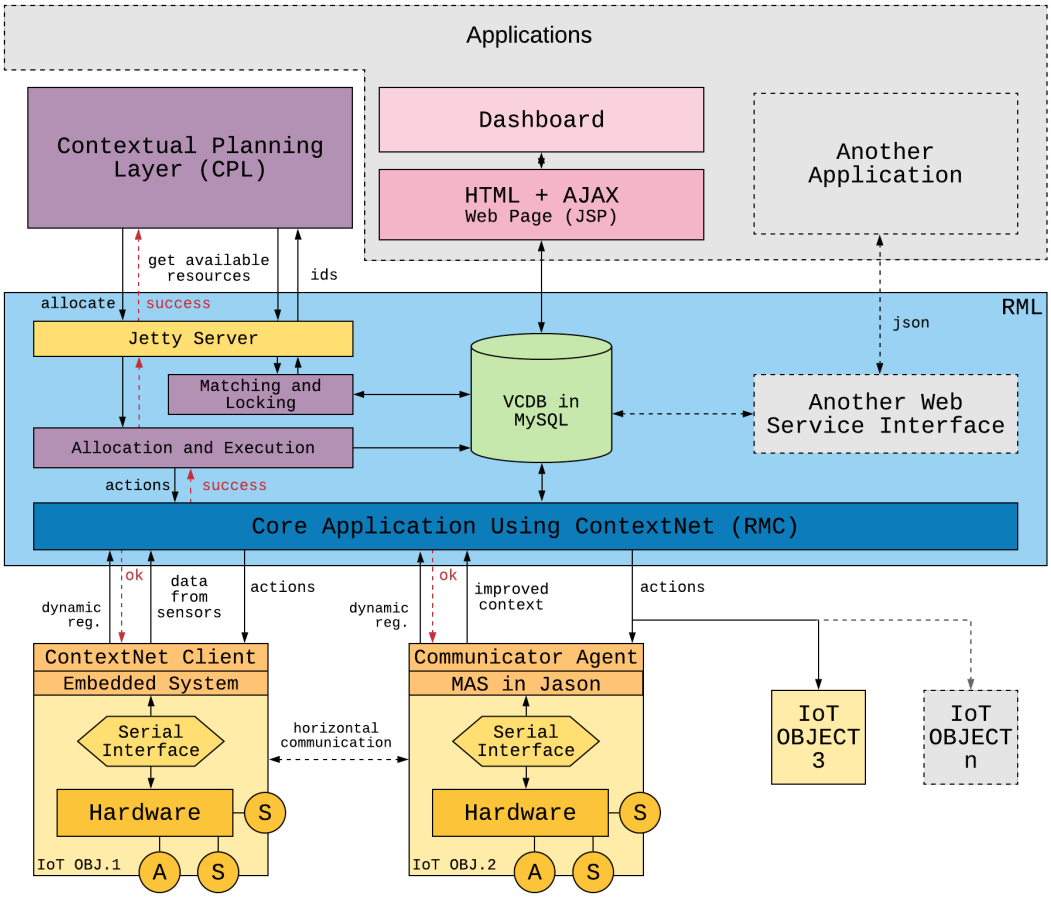

Figure 5.6: An overview of the proposed architecture.

The RML is a server instance of the ContextNet running a core system responsible for the virtualization of environments, IoT Objects, and their available resources. The IoT Objects registers at the RML informing their resources and environment where they are situated. This process starts when the core system receives all the information of the

IoT Object. Then, it registers the IoT Object at the system in the informed environment. Afterward, it sends back a message to the IoT Object authorizing the beginning of sending data from its resources. Once the IoT Objects start sending data, the core system registers in VCDB (Figure 4.1) the values that have changed since the last message received, if the data have not changed, there is no update of resources' values in the VCBD.

For every IoT Object, some resources can be sensors or actuators. In the case of sensors, if it is the first time that a new value of a resource is received, the system inserts this new value at the VCDB. Otherwise, if the value has changed since the last data reception, this new value is updated in VCBD. In the case of actuators, there is no data to be stored, but it is kept the information about the availability of the resource. For example, the system informs if a specific actuator is being used or it is free for executing commands. The core system deals with commands requisitions coming from the Application Layer (using web systems and services) to be sent and executed at IoT Objects. The Application layer does not know technical details of hardware or even in which IoT Object the command will be executed. The commands are specific for a resource, and the core system avoids duplicated resources and commands names. Besides, it also redirects a received command to the respective IoT Object by sending a message using ContextNet, since it keeps the identification of each IoT Object registered in the system.

The IoST considers a model where resources are part of devices — IoT objects in IoST — situated in existing environments in the architecture. For instance, the environments have to be manually registered by the user to facilitate their management and for security issues. The RML's model is composed of the *ResourceManagement* class where the ContextNet server is instantiated. Besides, it hosts a list of *Environments* with their *Devices*, *Resources*, and *Commands* classes. This model is represented as a class diagram (Figure 5.7) shared between the three layers. The classes that can be found in our solution are described as follows:

- **Action**: the action that is executed at the low-level hardware in a device. Every command sent by the Client and Cloud layers becomes an action in the Device layer.

- **Command**: it is the representation of commands available to be executed if the resource is an actuator. Sensors do not have commands because they are data providers.

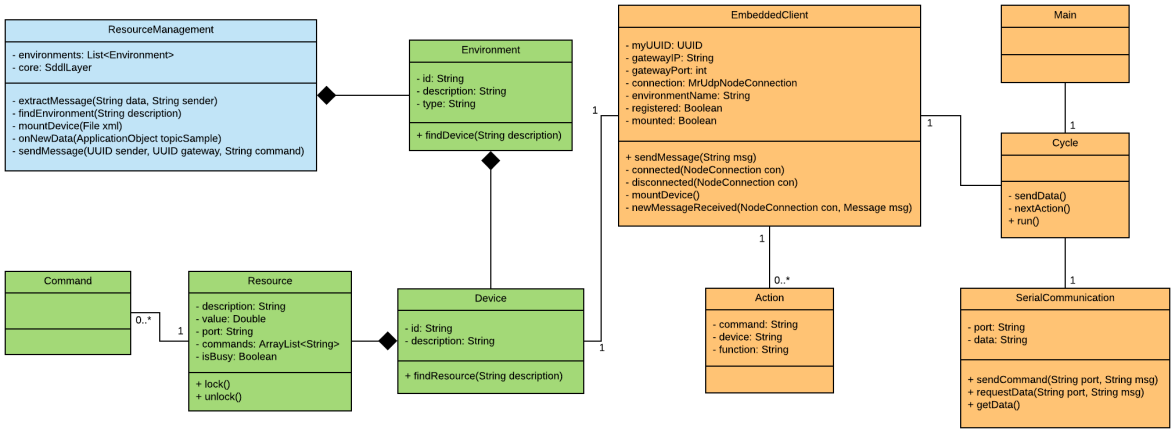

Figure 5.7: The class diagram of the overall architecture comprising the Client, Cloud and Device layers.

- **Cycle**: it is the functioning cycle of the embedded system hosted in devices. It is responsible for synchronizing the device activities of sending data and executing actions.

- **Device**: it is the device representation used in the Client, Cloud, and Device layers. It keeps the identification, name, and description of a device, and it is composed of resources.

- **Embedded Client**: it is the ContextNet client instance responsible for receiving messages from the Cloud layer and other clients, and sending the data from sensors to the RML. It also maps the components of the configuration file into its respective components.

- **Environment**: the virtual representation of environments in all layers. Each environment is composed of devices.

- **Main**: the main class that starts a device.

- **Resource**: it represents both sensors and actuators in all layers. The resources keep information about the serial port where the resource is connected, the available commands to be executed, and the availability of the resource.

- **Resource Management**: it is the main class of the cloud layer, and it is a server instance of the ContextNet. It keeps a list of environments mapped and the devices registered for each one. It is responsible for the process of registering and updating

devices and resources. It also exposes the web servers and the interfaces to the database.

- **Serial Communication**: the serial interface between the micro-controller and the system hosted at devices.

Moreover, the RML deals with requisitions of IoST Clients that are Services. Depending on which service is being executed, the processing of the request will be different. Hence, the core system of RML deals with processing registering IoT Objects, updating and keeping data, and data access requests from IoST Clients. In the next sections, the interface service with Contextual Planning Layer where agents can interact with the RML is described and also the Management Dashboard for aiding IoST Users to manage their IoT Objects is described.

5.2.1 Contextual Planning Layer as IoST Client

As stated before, web services can be exposed to extending the functionalities of the RML. They facilitate how IoT Clients access the RML for retrieving data or requesting any action execution, for example. A Web service is an interoperable interface using the internet, which is used for machine-to-machine communication independently from technologies such as devices, hardware, and programming languages. A Representational State Transfer (REST) is a representational architecture that allows the creation of well-defined interfaces [73]. Hence, for creating RESTful web services, it is used as an embedded servlet container and a web server named Jetty [119]. The available web services in the RML is a requisition service for contextual planning agents for getting information, allocating, and using resources exposed by IoT Objects in the IoST.

Agents in the Contextual Planning Layer (CPL) need to consider resources as part of their planning mechanisms. Hence, CPL accesses the RML using RESTful web services to obtain information about environments and resources available for the execution of plans. Based on the type of resources needed by the CPL, the RML performs a matching process answering it with all available resources identifications that matched the requisition. At this time, there is also a locking process for avoiding conflicts in executing actions when two or more agents try to use the same resource. In the case of sensors, there is no need for locking since one or more agents can consume the data without interfering the agents' planning mechanism or the IoT Object functioning. However, every time an agent needs to use a specific actuator, the core system locks this resource until the agent informs that

is no longer using it, the action was performed, or a timeout boundary is reached. During this process, the locked resource is unavailable for all IoT Clients.

The process starts by the CPL requesting the availability of specific resources by sending to the RML a JSON file with all resources that agents need in an environment for executing their plans. The RML answer it with another JSON file containing a list with the identification of the environments able of attending the CPL's request and it blocks the specific resources and environments related until CPL decides which one will be effectively used. This process is named Matching and Locking, and it is done by accessing the VCDB for verifying the availability of the resources. Afterward, the RML waits for a request for allocating a chosen environment — among the list sent previously — and it also waits for a list of actions to be executed. If this request takes so long to arrive at RML, a timeout process is performed, unlocking all environments and resources from the initial request. The timeout process waits for one minute until it releases all locked resources.

Otherwise, the CPL sends the identification of which environment and resources to allocate. This process unlocks all environments and resources from the initial request except the one chosen and allocated. Then, the actions are sent to be executed by IoT Objects in a process named Allocation and Execution. When actions need to be executed, the Allocation and Execution step sends the received actions to RMC, which is the core system with the server instance of ContextNet, that redirects each action to the respective IoT Objects. Then, when the actions were sent to the IoT Objects to be executed, all resources are unlocked, and the CPL is informed.

5.2.2 Management Dashboard as IoST Client

The idea behind the Applications layer is to provide a layer capable of showing environments' resources in any platform such as web pages, web services, or mobile applications. Besides, it is responsible for providing some basic mechanisms that are not available in previous layers such as environments creation for example. The Management Dashboard is represented as a web page for showing all the IoT Objects related to environments and some essential functions for interacting with their resources. An IoST Client is able of creating environments to virtually host his IoT Objects and to expose them to be consumed by other IoST Clients. Besides, the actuators have commands that users can activate by interacting with the Management Dashboard. The technologies employed are relational databases, Java web pages, and Ajax.

The Management Dashboard employs a web system capable of managing environments, showing their available resources, and it sends actions to IoT Objects. This web system allows the IoST Clients who own IoT objects to interact with their specific resources. Besides, this dashboard works as a center of control where the Users can configure if the data is public or private and to send actions privately. For now, other Clients are just able to visualize the public data, and they are not allowed to send actions to IoT Objects.

One of the benefits that can be explored by using web systems is that users can choose to follow some of the resources without the need to access the environments every time that he needs to get those values. By adopting a publish and subscribe mechanism, users can have access to resources' values always that the core system perceives a change in the RML. Moreover, users could set basic rules such as defining the desired value to be announced on a mobile phone when reached, for example. Since these are simple applications that use well-established technologies, these mechanisms are left as future works.

In time, all existing layers can co-exist without interfering in each other since all functions are managed by the RML, which centralizes and executes the requisitions from other layers one by one. Hence, there are no possibilities of more than one request to be executed at the same time. However, in practice, this issue is not perceptible by IoST Users. Besides, some of the IoST Clients access the RML's database — the VCDB — directly, since they need to visualize the data stored of IoT Objects, which does not interfere in RML's core system at all.

Chapter 6

Experimentation

In this chapter, it is presented some experiments and a case study evaluation in the assisted environment domain using the proposed RMA exploiting a software engineering-based approach. Case studies are employed because they are suitable for the evaluation of software engineering methods involving development, operation, maintenance, and its artifacts [129].

6.1 Case Study

First, the IoST must be provided to allow the connection of devices. For this, a server with an instance of ContextNet middleware was configured as well as all the network requirements needed. ContextNet is responsible for the connectivity, communicability, reliability, and scalability of the IoST. A Smart Thing is equipped with controllers, sensors, and actuators connected to any computational platform with an operational system and sufficient processing power for hosting a MAS. Tiny computers are commonly used since they are small, hold serial ports, and on-board communication technologies. So, we employed in the Smart Things ATMEGA328 controllers connected with temperature and luminosity sensors and power plug where electricity can be turned on or off, and a Raspberry Pi Zero for hosting the MAS. In the MAS was employed one *Communicator* agent per device, one *Interface* agent per controller, and one *Traditional* agent for aiding the other ones in the communication process.

In order to test the functional attributes of Smart Things, some devices were prepared to attend our proof-of-concept. The autonomy of devices was tested by verifying if sensors values were still being gathered when the server was disconnected. So, we assembled one Smart Thing responsible for gathering data from a temperature sensor and depending

on the temperature measured, it should turn on or off the plug where an air-conditioner is connected. The Smart Thing should connect to the ContextNet and after that, the server was disconnected. We repeated this ten times and the Smart Thing was able to perform autonomously in all opportunities. Once the server is available, the connectivity and communicability were possible to be tested. Thus, we tested the behavior of Smart Things by sending messages with data from the luminosity sensor of one Smart Thing to another one. This former one is responsible for executing a turn on action in its light actuator based on the value received. There were no problems in the execution of these tests.

The context-awareness was tested using the former test but changing the information that was sent. Instead of sending just the raw information from luminosity sensor, the Smart Thing was able to gather and process this information as *dark* or *bright* and it sent this context for the second Smart Thing to execute the turn-on action. The heterogeneity was tested by changing the controller from ATMEGA328 to Galileo Gen 2 in the first Smart Thing. Again, it was not identified any problems during the execution and everything went well. The objective of this proof-of-concept is to highlight that it is possible to employ MAS in embedded systems to provide autonomous and pro-active devices that are not dependable on central services to keep reasoning. Besides, the IoST extends the Smart Thing reach, providing a layer with connectivity, scalability, and communication. These tests aim to validate the main functionalities and the communication between the layers of the approach.

Afterwards, a scenario was specified to show some of the behaviors of the RML. The scenario appears in a hypothetical Smart City where the government has access to a hospital where the RMA is available. Some rooms in the hospital building have IoT Objects for controlling the temperature and luminosity (as sensors), and light lamps of the room (as actuators), and other IoT Objects for measuring some of the patients' information such as heartbeat frequency. Therefore, every room in the hospital endowed with IoT Objects is considered an environment in the RMA and its IoT Objects' resources are available as a service for the board of directors, government and everyone interested in them. It is important to remark that, even in this hypothetical scenario, there is no real personal contact of patients available.

Based on this, two IoT Objects were prepared for a room named `Room 403`. Both IoT Objects use a Raspberry Pi Zero connected to an Arduino board. The first IoT Object uses temperature and luminosity sensors for the basic sensing of the room. The

second one is an IoT Object with a light lamp connected to the Arduino working as an actuator and informing if the lamps are on or off. Besides, simulated virtual IoT Objects stress RMC functioning. For this, the serial interface between the embedded system and the hardware were disabled, and several simulated resources are available for each virtual IoT Object, which sends random data to the RML. So, one IoT Object for monitoring the heartbeat frequency of a patient and IoT Objects identical to the real ones above were simulated in each room. In general, 20 environments were prepared where the environment `Room 403` has two real IoT Objects and one simulated, and the other 19 have three simulated resources. The case study approach is divided into four steps: case study design, preparation for data collecting, the data collecting, and data analysis. The following Table 6.1 shows the details and aim of the descriptive case study.

Table 6.1: The Case Study Design

Design	**Description**
Objective	A descriptive analysis of the behavior of the RMA functioning.
Case	The asynchronous process of transferring information from IoT Objects to the RML to be consumed by clients using web solutions.
Questions	Does the IoT Object connects correctly to the RML? Does the communication process between all layers works? Is the Application layer showing the correct data?
Method	Qualitative data analysis using negative case analysis and observation method.

Some tests were conducted trying to deny the research questions above. The preparation for the data collecting consisted in store both dynamic registration at the RMC and the answer that IoT Objects receive before starting sending data from sensors. Afterward, it is verified if these data arrives at all layers correctly by analyzing the transferring process between hardware and IoT Object, IoT Object and RML, and RML and applications. A string of data that comes out from the hardware is collected and compared if the data read arrived correctly at the embedded system. Between Device and the Resource Management layers, all IoT Objects should keep sending data to be stored at the VCDB in the RMC. Finally, between RML and Application layer, these same data should be read by clients when data update occurs.

The data collecting and analysis were performed in an arithmetic progression from 1 to 20 devices. Firstly, the server instance was running properly to verify the effectiveness of RMA, and then it was disabled. Once there is no server instance available, they should not send data. Then, the data should be adequately stored and read by the Resource Management and Application layers. Table 6.2 shows the resumed results from tests.

Table 6.2: Data Collection and Analysis

Test	Description	Hit (%)
Hardware	Data is correctly transferred to the embedded system.	100
Connection with Server	IoT Object registered at RMC and registered message received.	100
Connection without Server	IoT Object registered at RMC and registered message received.	0
Data Updated and Stored	Data correctly stored at VCDB.	100
Data Read	Data correctly read by applications.	100

The communication between hardware and the embedded system is done using a serial interface, which guarantees no losses in the data transferring. None errors were observed in this process. As expected, when the server instance is disabled, it is observed that devices try to connect to the RML, but there is no response from the server, and the IoT Object did not send any data. Otherwise, the IoT Object is registered and receives a confirmation to start sending data to the RML. All IoT Objects work properly considering an available server instance.

The most recent information available coming from IoT Objects is updated in the VCDB. Considering that the VCDB uses a relational database, there are no big deals in this process not even in the visualization of the IoT Objects by the Application layer. This case study focused on observing the communication and the correctness of data flowing through the architecture. More experiments focusing on performance and a proper formalization were left for future efforts.

6.2 Performance Tests

In this section, we present some performance tests and discussions regarding some of the IoST components. These tests aim to provide an understanding of the behavior of the architecture during its execution. It is essential to verify the IoST processing capacity when it deals with high numbers of IoT Objects. The cost of processing messages in RML, the connection time of IoT Objects, and the time of sending messages will be explored to determine whether or not to use the architecture. All tests were executed in a regular machine, with an i5 processor of 1.7 GHz and 8Gb of RAM. Besides, we also present some experiments regarding requisitions from CPL to the RML. For this, the first

test considered how long it takes to IoT Objects to connect at RML. All IoT Objects must connect to the RML to allow their data to be shared and consumed by services and other clients. The connection to the RML is the first step that must be accomplished by the IoT Objects before they start to send data. However, while a connection request is sent, the RML can be treating other requisitions such as data processing, connection and disconnection, and actions messages coming from several IoT Objects. Hence, we simulated an AmI system with 30 IoT Objects and evaluated the time response for a connection request. We captured the moment when the request is sent, and when each IoT Object received the connection feedback. We also configured the interval of sending messages with 1 second, and after that, up to 5 seconds. The results can be seen in plotted graphics in Figure 6.1.

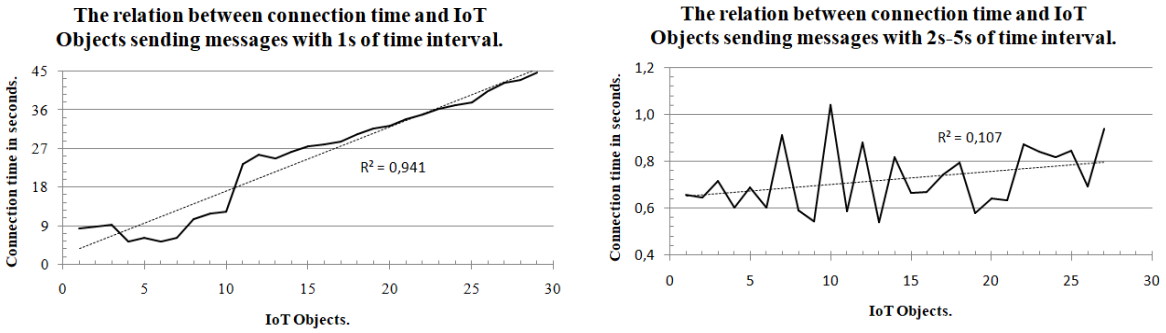

Figure 6.1: The relation between the connection time and IoT Objects considering the time interval of sending messages from 1 to up 5 seconds, and 30 IoT Objects.

The first graphic shows that the connection time increases as the number of devices grow. The determination coefficient corroborates the dependency between the connection time in seconds and the number of IoT Objects. It is expected that as IoT Objects register themselves, they start to send messages from one to one second, in the first case. Hence, as the number of IoT Objects grows, the number of messages that should be processed by the RML grows as well. Then, the RML deals with the message that arrives first independently from which type it is. IoT Objects use the ContextNet as a communication channel, and this SDDL middleware handles all messages. The RML does not have access to how ContextNet stores and sorts incoming messages. If so, a priority mechanism for such type of messages could be provided and tested.

Nevertheless, we explored varying the interval of sending messages from 2 to 5 seconds to verify if the behavior holds. The results show that the connection time starts to increase slowly as new IoT Objects connected to the RML. However, the connection time decreased for a few IoT Objects connected. If the number of IoT Objects necessary for running an

AmI system is too large, eventually the connection time will grow again. The connection time does not interfere in the RML processing or capacity, but it adds an undesirable waiting time that IoT Objects have to wait for being part of the system. Then, we run a test considering a dedicated service for connecting IoT Objects apart from sending messages to compare the performance and the response time. The results can be seen in Figure 6.2.

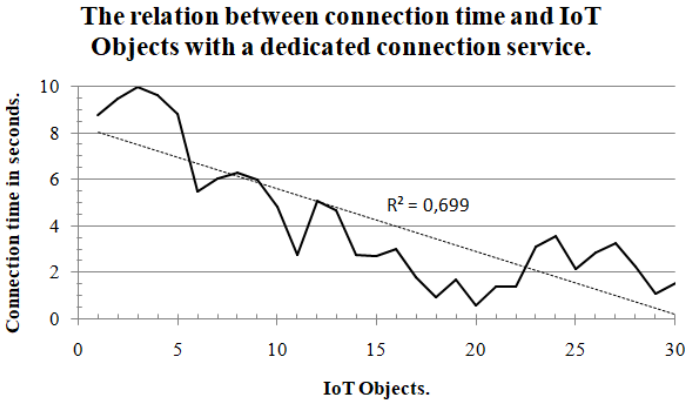

Figure 6.2: The relation between the connection time and IoT Objects considering a dedicated connection service, and 30 IoT Objects.

In this case, the connection time cost reduces and tends to stabilize. This variability can be corroborated by measuring in seconds the standard deviation and average of the connection time of all IoT Objects for this test. The average is 4.13 seconds, and the standard deviation is 2.84 seconds. Comparing the same metrics for the previous tests, when the interval time of sending messages is 1 second, we obtain an average of 24.57 seconds and a standard deviation of 13.09 seconds. In the other case, considering the interval time of 5 seconds, the average is 0.73 second, and the standard deviation is 0.13 seconds. The results for the time interval of 5 seconds overcame the one with a dedicated service for connection of IoT Objects. It was expected the opposite. However, we increased the number of IoT Objects to 50 to analyze if these results still hold. Figure 6.3 shows the results for the interval time of 1 and 5 seconds with 50 IoT Objects.

As expected and observed from past tests, when the number of IoT Objects increases the connection time also increases. However, when the interval time of sending messages is 5 seconds, the connection time grows more than the similar one with 30 IoT Objects, and it starts to increase from 30 to 50 IoT Objects slowly. It is essential to understand that the time is growing because RML has to process every message that arrives. Hence, there is a processing cost associated with the RML for every connection request. Understand

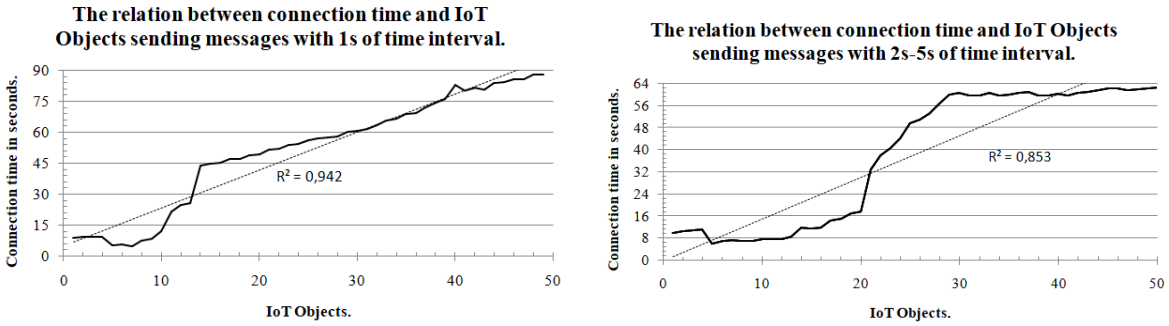

Figure 6.3: The relation between the connection time and IoT Objects considering the time interval of sending messages from 1 to up 5 seconds, and 50 IoT Objects.

the impact that this process cost has in the RML will help to set some limitations and approaches to deal with a large number of IoT Objects. For this, the next test considered the processing cost in the RML associated to the connection time spent for each IoT Object. Then, it will make it possible to observe the impact that the RML effectively has in the connection process. Figure 6.4 shows the processing cost and the connection time with a dedicated service exclusive for connecting IoT Objects.

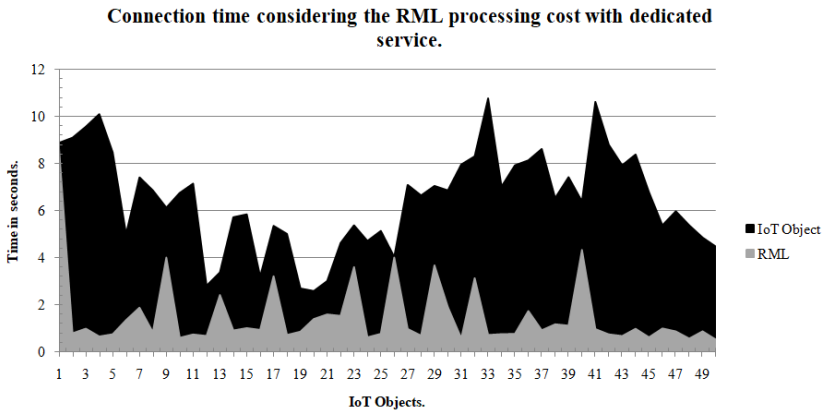

Figure 6.4: The relation between the connection time and IoT Objects considering the time interval of sending messages from 1 to up 5 seconds, 50 IoT Objects, and the associated RML processing cost.

It is possible to see that the processing cost of the RML is not the main cost in the connection process of an IoT Object. Other costs might be associated such as the network latency for example. However, it is essential to identify the capacity of RML and the cost associated with it for every message that it has to process since the RML centralizes all messages of IoT Objects connected to it. Then, we tested the processing capacity of RML by sending plenty of messages from the 50 IoT Objects. Figure 6.5 shows the graphic related to the referred test.

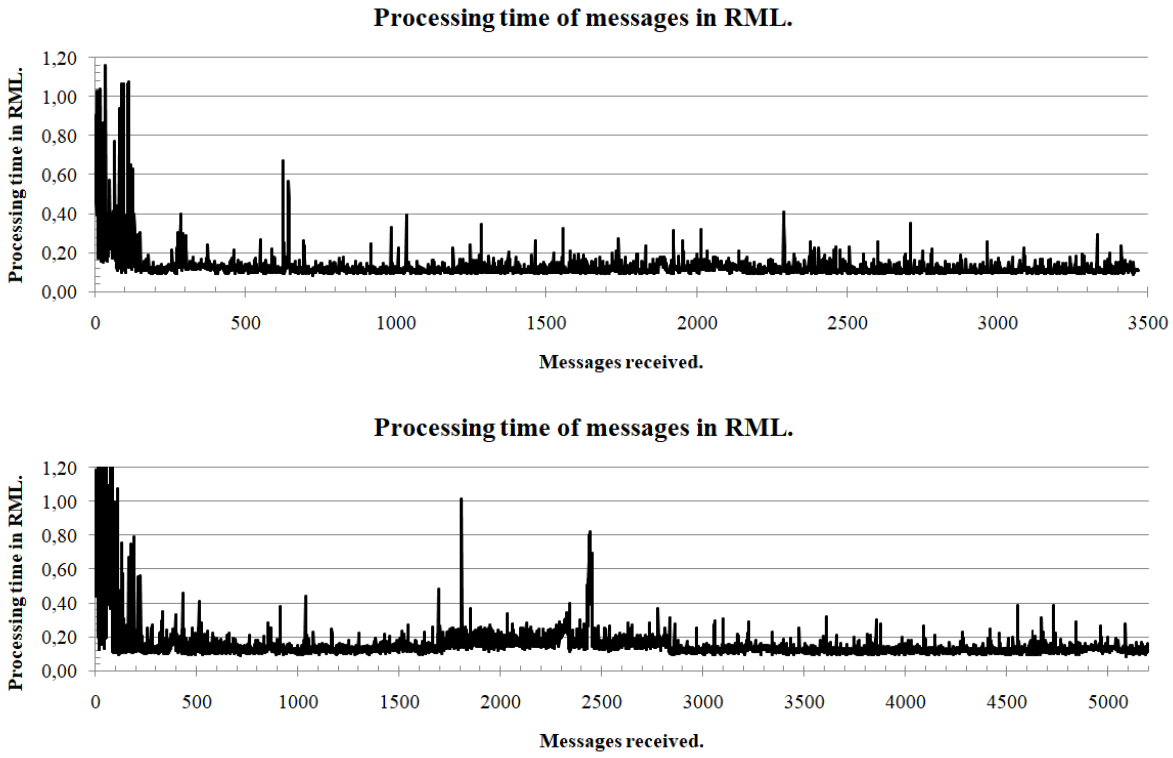

Figure 6.5: The relation between the processing time in RML and the number of messages received.

We configured two scenarios where 50 IoT Objects sent messages to the RML. In the first case, the architecture processed near to 3500 messages, and in the second one, it processed about 5000 messages. The interval time of messages sent is irrelevant for the processing that occurs in the core application. It is possible to observe that there is a higher cost associated with the connection of IoT Objects in the first 100 messages. After that, there is no significant variation in message processing. The average time of processing messages is 0.13 seconds, with a standard deviation time of 0.07 seconds.

Considering the average time of processing one message in the RML, it is possible to define an upper limit of IoT Objects — if every IoT Object sends exactly one message — that can be attended per second — or minute — using the hardware configuration for these tests. Approximately, it is possible to attend eight messages per second and 480 messages per minute in the tested version of the RML. However, the hardware is too limited for this purpose, and several improvements can be applied in the RML programming to reduce the processing time further. Anyhow, there will always be a processing limitation when considering hardware technologies that will define the applicability of any computer application, including the RML. This limitation creates a queue of messages

in the ContextNet that still need to be sent to be processed by the RML. Moreover, if the amount of queued messages grows more than the capacity of the RML of processing messages, the system becomes infeasible.

To overcome this issue, several instances of the RML could run in different servers. All components of RML can be replicated in other servers since the RML does not store any data from IoT Objects. The VCDB is responsible for storing all the information needed to the system works properly, and it can be accessed by other services as well. In this way, the network infrastructure can grow whenever new IoT Devices enter the IoST, or new virtual environments are created to host IoT Devices, and the current architecture does not support new additions. However, in an *ad hoc* application focusing on a building where the number of IoT Devices will not grow significantly during the time, the IoST can deliver proper ambients and resources management.

6.3 CPL-RML Experiments

Here, it is presented some experiments considering the interaction between the RML and CPL as a proof-of-concept for this service. These tests were conducted in collaboration with the LIP6 laboratory at the Université Pierre et Marie Curie (Sorbonne UPMC). The proof-of-concept was validated in conjunction with an application scenario. It is important to remark that both layers were deployed to the Web as RESTful services. The objective of these tests was to observe if the architecture would be able to properly deal with requests from multiple agents and multiple resources at the same time. The agent plans used in these experiments will not be detailed because they are part of the planning mechanism provided by the CPL. The scenario contains multiple agents, with multiple plans depending on common resources.

The proof-of-concept was evaluated with a simple scenario involving multiple agents. Each agent possessed different plans, and some of them depended on physical resources. We also considered different sets of resources grouped into five different environment abstractions. Each of these environments possessed a limited capacity and a subset of the following resources: thermostat, camera, illumination controls, and projector.

For this thesis, the scenario was executed in two different environmental conditions, considering the presence of available resources. In the first context (1), each agent was given a set of plans that did not contain resource-dependent elementary plans (in CPL). In this context, we expected that only the plans depending on other constraints than

resources would be unfeasible. This execution establishes a baseline for the next execution, where resource dependency was added to the agent's plans used in the previous execution. The resources were not necessarily available in the environment. Thus, it is expected that some of the otherwise feasible plans would be marked as unfeasible due to resource constraints. Six different sets of environment configurations were used as requirements (Table 6.3). Each environment configuration was designed to match none, one or multiple of the environment abstractions in RML.

Table 6.3: The environment configuration.

Configuration	Capacity	Resources	Matches
1	20	thermostat, light controls	1, 4
2	40	temperature	1
3	60	light controls	-
4	20	camera	3
5	20	light controls	1, 2, 4, 5
6	20	temperature, computer	-

Once defined the environment configurations, the agent's plans were executed to verify the resource allocation mechanism and its impact on plan feasibility. The test has objectives into (i) availability (checking if a given environment set was available upon request), (ii) concurrency (checking if the resource limit was observed, allowing plans depending on equivalent environment configurations to be executed accordingly) and (iii) multiple dependency (checking if plans depending on multiple environments could be fully or partially executed, and how the RML would handle such situations). An example of test cases executed according to these directives is shown in Table 6.3. Each case is related to a given objective, the configuration sequence used (environments sequentially required by agent plans), and the expected allocation of the available environments. Both configuration sequences and the allocated environments refer to the tables previously shown. The operator "+" refers to multiple elements in the same requisition.

Table 6.4: The CPL-RML tests.

	Objective	Config. Sequence	Alloc. Env.
1	Availability	3	-
2	Concurrency	4, 4, 1, 1, 1	3, -, 1, 4, -
3	Multiple Dependency	2+4, 2+4	1+3, -

Chapter 7

Final Remarks

It was presented a three-layer architecture for exposing IoT Objects that are able of connecting and registering their resources (sensors and actuators) in a middle layer, which makes all the public data available to be accessed by client applications. Hence, users can create real environments with their own IoT Objects, and they could share it publicly using the RMA. This architecture was named Internet of Smart Things (IoST), and it comprises three layers: Device, Resource Management, and Application.

In the Device layer, the architecture employs Things and Smart Things, autonomous IoT Objects endowed with hardware resources and built using hardware platforms enhanced with wi-fi connections, and they use a serial interface for communicating with microcontrollers. These technologies provide the necessary autonomy, heterogeneity of hardware employed, and communicability to the RML. Besides, Smart Things uses MAS as the embedded system for controlling the Smart Things functionalities and reasoning.

The RML employs different technologies: the ContextNet middleware provides client and server instances for both IoT Objects and the RML respectively, and it is used in the architecture because of the middleware guarantees connectivity, communicability, reliability, and scalability, using an industrial market standard protocol. The Application layer offers solutions for the visualization of IoT objects that are situated in environments that can be managed by the IoT Objects' owner. Besides, it is possible to subscribe individually to some resources that one might be interested in accessing. Moreover, the IoST provides an architecture for exposing IoT Objects to be consumed by clients that do not have access to these kinds of resources either for the cost or complexity of creating from scratch an architecture for that purpose. One of the available application is directed to CPL agents. The CPL adds the resource dependability in how to generate feasible plans using the RML. Initially, the CPL did not use physically distributed resources as

part of its planning mechanism.

In this thesis, it was also presented an extension of the agent-oriented framework Jason to allow embedded MAS in Smart Things to communicate with other agents hosted and embedded in different Smart Things. This extension provides a specific and new kind of agent that can communicate with other agents using the *ContextNet*. In this type of agents, the middleware is part of the reasoning cycle of the agent, which uses internal actions for sending illocutionary messages to agents with the same ability and hosted in a different MAS. Using *ContextNet* and **Communicators** agents, it is possible to an embedded MAS to enter or leave an intelligent ambient or AmI system by connecting and disconnecting from the IoST to interact with other devices without concerning about modifications in the code of the MAS.

7.1 Published Works

This thesis generated several works that were published during the development of the IoST. As the architecture is divided into three layers — Device, Cloud and Service — there are published works considering technological issues for allowing the development of embedded MAS systems in IoT devices, and how to use these devices in an architecture exploiting advantages from the agent approach, AmI and IoT together to aid developers and users in how to provide and use their services. In order to facilitate in visualizing the published and submitted works, Figure 7.1 shows a timetable and dependency of all published works related to the development of this thesis.

Starting from Smart Things, employ embedded MAS is necessary to provide autonomy and pro-activity in these IoT devices. However, to pursue this goal, it was necessary to allow the development of embedded agents and MAS for guaranteeing the management of hardware platforms by such systems. Hence, a hardware interface was proposed to transfer information between hardware and software in 2015. The work entitled **A Robotic-agent Platform For Embedding Software Agents using Raspberry Pi and Arduino Boards** [63] was published in the *Workshop-Escola de Sistemas de Agentes, Seus Ambientes e Aplicações* in 2015. The same idea was extended for another type of controller in the work **A Middleware for Using PIC Microcontrollers and Jason Framework for Programming Multi-Agent Systems** [50] published in the *I Workshop de Pesquisa em Computação dos Campos Gerais (WPCCG)* in 2016. The hardware interface allowed the development of a new kind of agent able to controlling hardware directly from

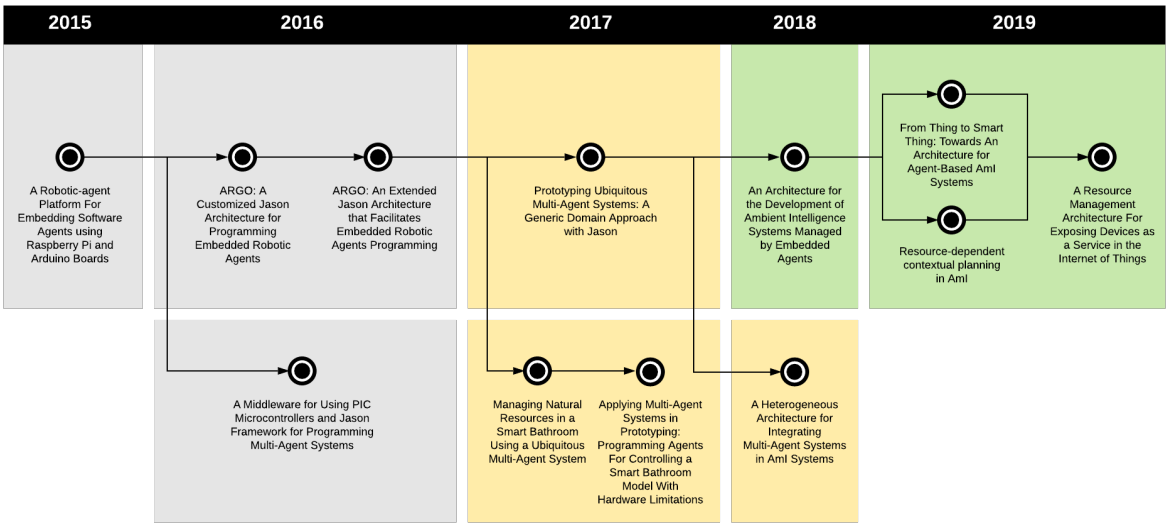

Figure 7.1: The timeline of the published works related to this thesis: the gray squares show the works developed at the beginning of the Ph.D. The yellow squares show the works developed after professor Viterbo became the advisor. The green squares show the works after professor Amal became the co-advisor.

its reasoning cycle. It was described in details in **ARGO: A Customized Jason Architecture for Programming Embedded Robotic Agents** [90] it was selected to be extended in **ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming** [89] published in the *Lecture Notes in Computer Science, v. 10093* both in 2016.

Even though they are not the focus of this thesis and they were developed before the conception of the IoST architecture, they are essential to provide the basis of Smart Things, and from this basis, it was possible to explore some abilities. The heterogeneity of controllers and the capability of being applied in any domain was exploited in **A Heterogeneous Architecture for Integrating Multi-Agent Systems in AmI Systems** [40] in *The 30th International Conference on Software Engineering and Knowledge Engineering* in 2018 and **Prototyping Ubiquitous Multi-Agent Systems: A Generic Domain Approach with Jason** [91] published in the *Lecture Notes in Computer Science, v. 10349* in the proceedings of *International Conference on Practical Applications of Agents and Multi-Agent Systems* in 2017.

Besides, real prototypes were developed using the embedded approach such the one described in **Managing Natural Resources in a Smart Bathroom Using a Ubiquitous Multi-Agent System.** [14] published in the *XI Workshop-Escola de Sistemas de Agentes, seus Ambientes e Aplicações* and selected as one of the best papers and extended

in **Applying Multi-Agent Systems in Prototyping: Programming Agents For Controlling a Smart Bathroom Model With Hardware Limitations** [69] in the *Revista Júnior de Iniciação Científica em Exatas e Engenharia* in 2017.

The architecture considering the Cloud and Device layers where the ability of communicability and connectivity were inserted was explored in the **An Architecture for the Development of Ambient Intelligence Systems Managed by Embedded Agents** [88] published in the *30th International Conference on Software Engineering and Knowledge Engineering* in 2018, and in **From Thing to Smart Thing: Towards An Architecture for Agent-Based AmI Systems** [92] published in the *Proceedings of the Agent and Multi-Agent Systems: Technology and Applications, 13th KES International Conference* in 2019.

The RML integrated to the CPL were proposed in **Resource-dependent contextual planning in AmI** [20] published in the journal *Procedia Computer Science, v. 151* in 2019. The work entitled **A Resource Management Architecture For Exposing Devices as a Service in the Internet of Things** [87] was published in the *The 31st International Conference on Software Engineering and Knowledge Engineering* in 2019, and it was selected to be extended in the upcoming issue of the *International Journal of Software Engineering and Knowledge Engineering* also in 2019.

7.2 Limitations

At this point, it is essential to remark some of the identified limitations of the presented approach. The IoST is dependent on available gateways with a single RML instance installed, and it is not prepared to be a distributed layer. It centralizes the flow of data to a single point in the solution obligating the available IoT Objects to connect to it. Engineering depends on a lot of effort from the designer of the IoST Object since he needs to mount and configure all resources manually for the self-registering mechanism works adequately.

The MAS of Smart Things must employ only one type of Communicator agent, which can communicate and connect in the IoST. There are no practical limitations in using more than one Communicator agent, but it is employed to identify the Smart Thing uniquely in the IoST. If two or more Communicator agents were employed, the RML would not deal with the ambiguity of which agent will be responsible for interacting with the IoST. However, the Smart Thing will still be functional, and both of them could perform the

communicability with other IoT Objects.

Another issue regarding the IoST Objects is that they need to know each other identification if they want to communicate. Even though the agents can introduce to each other and exchange plans (including introducing plans), it is not possible to identify newcomer IoT Objects without being prepared for that, and it depends on the IoST Designer to provide such mechanisms. However, it is possible to adapt the RML to broadcast a list of active IoST Objects to the interested Smart Thing.

7.3 Future Works

Concerning the agent approach, social organizations of Smart Things could represent an advance in how to achieve collective goals in IoT. Social organizations consider groups of agents organized for achieving common goals, with well-defined missions based on roles that an agent could play depending on what is expected from it on the social organization, associated mission or collective goal [55]. Distributed approaches using simulated agents have been successfully employed during the past years exploring agent organizations and social interaction among groups. However, the groups are composed of agents treated as individual entities from a pre-existing MAS. Since a Smart Thing is a single and autonomous entity controlled by MAS, it can be itself part of a group along with other Smart Thing to perform a collective behavior in the IoST. For applying social organizations of Smart Things, the ContextNet already deals with a group of clients. This characteristic could be explored or extended to allow the creation of organizations or society of Smart Things.

From the data flowing in the architecture, it is interesting to explore machine and stream learning technologies in this flow of data to identify patterns, behaviors, and situations in an AmI system. We also aim to explore the ability to use rules in the ContextNet middleware along with Things and Smart Things. Besides, we also aim to create a testbed for automation of experiments and resource sharing, in order to assist in the validation of new proposals involving IoT technologies and MAS.

In the engineering process of IoT Objects for the IoST, the designer of the device should program how data is mounted and captured by the microcontroller and sent to the embedded system. As future works, it is essential to create an automatized plug-and-play way of configuring the device in low-level. Besides, as microcontrollers are connected to the serial port of the tiny computer of the device, a similar process for the identification

of serial ports by the embedded system is also interesting. These issues will facilitate and automate in how the IoT Objects are programmed and mounted. The environment has to be set manually at the device's configuration file for security and control reasons. Nevertheless, it is possible to identify and register autonomously the environment based on access points or any other indoor localization system.

References

[1] Abelha, A.; Analide, C.; Machado, J.; Neves, J.; Santos, M.; Novais, P. Ambient intelligence and simulation in health care virtual scenarios. In *Establishing the Foundation of Collaborative Networks* (Boston, MA, 2007), L. M. Camarinha-Matos, H. Afsarmanesh, P. Novais, and C. Analide, Eds., Springer US, pp. 461–468.

[2] Abowd, G. D.; Dey, A. K.; Brown, P. J.; Davies, N.; Smith, M.; Steggles, P. Towards a better understanding of context and context-awareness. In *Handheld and Ubiquitous Computing* (Berlin, Heidelberg, 1999), H.-W. Gellersen, Ed., Springer Berlin Heidelberg, pp. 304–307.

[3] Andrade, J. P. B.; Oliveira, M.; Gonçalves, E. J. T.; Maia, M. E. F. Uma Abordagem com Sistemas Multiagentes para Controle Autônomo de Casas Inteligentes. In *XIII Encontro Nacional de Inteligência Artificial e Computacional (ENIAC)* (2016).

[4] Atzori, L.; Iera, A.; Morabito, G. The internet of things: A survey. *Computer networks 54*, 15 (2010), 2787–2805.

[5] Augusto, J. C.; Liu, J.; McCullagh, P.; Wang, H.; Yang, J.-B. Management of uncertainty and spatio-temporal aspects for monitoring and diagnosis in a smart home. *International Journal of Computational Intelligence Systems 1*, 4 (2008), 361–378.

[6] Barros, R. S.; Heringer, V. H.; Lazarin, N. M.; Pantoja, C. E.; Moraes, L. M. An agent-oriented ground vehicle's automation using Jason framework. In 6^{th} *International Conference on Agents and Artificial Intelligence* (2014), pp. 261–266.

[7] Bellifemine, F. L.; Caire, G.; Greenwood, D. *Developing multi-agent systems with JADE*, vol. 7. John Wiley & Sons, 2007.

[8] Benta, K.-I.; Hoszu, A.; Văcariu, L.; Creţ, O. Agent based smart house platform with affective control. In *Proceedings of the 2009 Euro American Conference on Telematics and Information Systems: New Opportunities to increase Digital Citizenship* (2009), ACM, p. 18.

[9] Boissier, O.; Bordini, R. H.; Hübner, J. F.; Ricci, A.; Santi, A. Multi-agent oriented programming with jacamo. *Science of Computer Programming 78*, 6 (2013), 747–761.

[10] Bordini, R. H.; Braubach, L.; Dastani, M.; El, A.; Seghrouchni, F.; Gomez-Sanz, J. J.; Leite, J.; Pokahr, A.; Ricci, A. A survey of programming languages and platforms for multi-agent systems, 2006.

[11] Bordini, R. H.; Hübner, J. F.; Wooldridge, M. *Programming Multi-Agent Systems in AgentSpeak using Jason.* John Wiley & Sons Ltd, 2007.

[12] Bosse, S. Mobile multi-agent systems for the internet-of-things and clouds using the javascript agent machine platform and machine learning as a service. In *Future Internet of Things and Cloud (FiCloud), 2016 IEEE 4th International Conference on* (2016), IEEE, pp. 244–253.

[13] Bosse, T.; Hoogendoorn, M.; Klein, M. C.; Treur, J. A component-based ambient agent model for assessment of driving behaviour. In *International Conference on Ubiquitous Intelligence and Computing* (2008), Springer, pp. 229–243.

[14] Brandao, F.; Nunes, P.; Jesus, V. S.; Pantoja, C. E.; Viterbo, J. Managing Natural Resources in a Smart Bathroom Using a Ubiquitous Multi-Agent System. In 11^{th} *Software Agents, Environments and Applications School* (2017).

[15] Bratman, M. E. *Intention, Plans and Practical Reasoning.* Cambridge Press, 1987.

[16] Busemeyer, J. R.; Diederich, A. *Cognitive modeling.* Sage, 2010.

[17] Busetta, P.; Kuflik, T.; Merzi, M.; Rossi, S. Service delivery in smart environments by implicit organizations. In *Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on* (2004), IEEE, pp. 356–363.

[18] Busetta, P.; Merzi, M.; Rossi, S.; Zancanaro, M. Group communication for real-time role coordination and ambient intelligence. In *International Workshop on Artificial Intelligence in Mobile Systems–UbiComp* (2003), vol. 3.

[19] Busetta, P.; Rönnquist, R.; Hodgson, A.; Lucas, A. Jack intelligent agents-components for intelligent agents in java. *AgentLink News Letter 2*, 1 (1999), 2–5.

[20] Casals, A.; Fallah-Seghrouchni, A. E.; Brandão, A. A. F.; Pantoja, C. E.; Viterbo, J. Resource-dependent contextual planning in ami. *Procedia Computer Science 151* (2019), 485 – 492.

[21] Chaouche, A.-C.; El Fallah Seghrouchni, A.; Ilié, J.-M.; Saïdouni, D. E. A formal approach for contextual planning management: Application to smart campus environment. In *Advances in Artificial Intelligence – IBERAMIA 2014* (Cham, 2014), A. L. Bazzan and K. Pichara, Eds., Springer International Publishing, pp. 791–803.

[22] Chaouche, A.-C.; Seghrouchni, A. E. F.; Ilié, J.-M.; Saïdouni, D. E. A higher-order agent model with contextual planning management for ambient systems. In *Transactions on Computational Collective Intelligence XVI.* Springer, 2014, pp. 146–169.

[23] Chebout, M. S.; Mokhati, F.; Badri, M.; Babahenini, M. C. Towards preventive control for open mas - an aspect-based approach. In *Proceedings of the 13th International Conference on Informatics in Control, Automation and Robotics - Volume 1: ICINCO,* (2016), SciTePress, pp. 269–274.

[24] Chen, B.; Cheng, H. H. A review of the applications of agent technology in traffic and transportation systems. *IEEE Transactions on Intelligent Transportation Systems 11*, 2 (2010), 485–497.

[25] Chen, B.; Cheng, H. H.; Palen, J. Mobile-c: a mobile agent platform for mobile c/c++ agents. *Software: Practice and Experience 36*, 15 (2006), 1711–1733.

[26] Chen, B.; Cheng, H. H.; Palen, J. Integrating mobile agent technology with multi-agent systems for distributed traffic detection and management systems. *Transportation Research Part C: Emerging Technologies 17*, 1 (2009), 1–10.

[27] Clark, K.; Robinson, P. Robotic agent programming in TeleoR. In *Robotics and Automation, 2015 IEEE International Conference on* (2015), pp. 5040–5047.

[28] Conte, G.; Morganti, G.; Perdon, A. M.; Scaradozzi, D. Multi-agent system theory for resource management in home automation systems. *Journal of Physical Agents 3*, 2 (2009), 15–19.

[29] Cook, D. J.; Augusto, J. C.; Jakkula, V. R. Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing 5*, 4 (2009), 277–298.

[30] Cook, D. J.; Youngblood, G. M.; Heierman III, E. O.; Gopalratnam, K.; Rao, S.; Litvin, A.; Khawaja, F. Mavhome: An agent-based smart home. In *PerCom* (2003), vol. 3, pp. 521–524.

[31] Cook, D. J.; Youngblood, M.; Das, S. K. A multi-agent approach to controlling a smart environment. *Designing smart homes 4008* (2006), 165–182.

[32] Corchado, J. M.; Bajo, J.; Abraham, A. Gerami: Improving healthcare delivery in geriatric residences. *IEEE Intelligent Systems 23*, 2 (2008).

[33] Corchado, J. M.; Bajo, J.; De Paz, Y.; Tapia, D. I. Intelligent environment for monitoring alzheimer patients, agent technology for health care. *Decision Support Systems 44*, 2 (2008), 382–396.

[34] Costantini, S.; Mostarda, L.; Tocchio, A.; Tsintza, P. Dalica: Agent-based ambient intelligence for cultural-heritage scenarios. *IEEE Intelligent Systems 23*, 2 (2008).

[35] Dastani, M.; de Boer, F.; Dignum, F.; Van Der Hoek, W.; Kroese, M.; Meyer, J.-J., et al. Programming the deliberation cycle of cognitive robots. In *Proc. of the 3rd International Cognitive Robotics Workshop* (2002).

[36] David, L.; Vasconcelos, R.; Alves, L.; André, R.; Endler, M. A dds-based middleware for scalable tracking, communication and collaboration of mobile nodes. *Journal of Internet Services and Applications 4*, 1 (2013), 16.

[37] de Jesus, V. S.; da Silva, Y. F. S.; Pantoja, C. E.; Samyn, L. M. Lubras dispositivo eletrônico para comunicação libras–língua portuguesa. *Mostra Nacional de Robótica (MNR)* (2016).

[38] de Jesus, V. S.; Manoel, F. C. P.; Pantoja, C. E.; Viterbo, J. Transporte de agentes cognitivos entre sma distintos inspirado nos princıpios de relaç oes ecológicas. In *Workshop-Escola de Sistemas de Agentes, seus Ambientes e apliCações—XII WESAAC* (2018), pp. 179–187.

[39] de Jesus, V. S.; Manoel, F. C. P. B.; Pantoja, C. E. Protocolo de interação entre sma embarcados bio-inspirado na relação de predatismo. In *13th Software Agents, Environments and Applications School (WESAAC)* (2019).

[40] de Jesus, V. S.; Manoel, F. C. P. B.; Pantoja, C. E.; Viterbo, J. A heterogeneous architecture for integrating multi-agent systems in ami systems. In *30th Software Engineering & Knowledge Engineering* (2018).

[41] de Jesus, V. S.; Pantoja, C. E. Explorando o transporte de agentes cognitivos entre sistemas multi-agentes distintos. In *II Workshop de Pesquisa em Computação dos Campos Gerais (WPCCG 2017)* (2017).

[42] de Jesus, V. S.; Samyn, L. M.; Manoel, F. C. P.; Pantoja, C. E. Lubras: Uma arquitetura de um dispositivo eletrônico para a comunicação libras-língua portuguesa utilizando o javino. In *I Workshop de Pesquisa em Computação dos Campos Gerais (WPCCG)* (2016).

[43] de Souza, B. J. O.; Endler, M. Coordinating movement within swarms of uavs through mobile networks. In *2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)* (2015), IEEE, pp. 154–159.

[44] Desai, P.; Sheth, A.; Anantharam, P. Semantic gateway as a service architecture for iot interoperability. In *Mobile Services (MS), 2015 IEEE International Conference on* (2015), IEEE, pp. 313–319.

[45] Dey, A. K. Context-aware computing: The cyberdesk project. In *Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments* (1998), pp. 51–54.

[46] Durfee, E. H. Distributed problem solving and planning. In *Multi-Agent Systems and Applications: 9th ECCAI Advanced Course, ACAI 2001 and Agent Link's 3rd European Agent Systems Summer School, EASSS 2001 Prague, Czech Republic, July 2–13, 2001 Selected Tutorial Papers* (Berlin, Heidelberg, 2001), M. Luck, V. Mařík, O. Štěpánková, and R. Trappl, Eds., Springer Berlin Heidelberg, pp. 118–149.

[47] Endler, M.; Baptista, G.; Silva, L.; Vasconcelos, R.; Malcher, M.; Pantoja, V.; Pinheiro, V.; Viterbo, J. Contextnet: context reasoning and sharing middleware for large-scale pervasive collaboration and social networking. In *Proceedings of the Workshop on Posters and Demos Track* (2011), ACM, p. 2.

[48] Endler, M.; Silva, F. S. E. Past, present and future of the contextnet iomt middleware. *OJIOT 4*, 1 (2018), 7–23.

[49] Gomes, B.; Muniz, L.; e Silva, F. J. d. S.; Ríos, L. E. T.; Endler, M. A comprehensive cloud-based iot software infrastructure for ambient assisted living. In *2015 International Conference on Cloud Technologies and Applications (CloudTech)* (2015), IEEE, pp. 1–8.

[50] Guinelli, J. V.; Pantoja, C. E. A middleware for using pic microcontrollers and jason framework for programming multi-agent systems. In *I Workshop de Pesquisa em Computação dos Campos Gerais (WPCCG)* (2016).

[51] Hagras, H.; Callaghan, V.; Colley, M.; Clarke, G.; Pounds-Cornish, A.; Duman, H. Creating an ambient-intelligence environment using embedded agents. *IEEE Intelligent Systems 19*, 6 (2004), 12–20.

[52] Hernández, M. E. P.; Reiff-Marganiec, S. Towards a software framework for the autonomous internet of things. In *Future Internet of Things and Cloud (FiCloud), 2016 IEEE 4th International Conference on* (2016), IEEE, pp. 220–227.

[53] Hindriks, K. V. Programming rational agents in GOAL. In *Multi-Agent Programming: Languages, Tools and Applications*, A. Seghrouchni, J. Dix, M. Dastani, and H. R. Bordini, Eds. Springer US, Boston, MA, 2009, pp. 119–157.

[54] Hindriks, K. V.; De Boer, F. S.; Van der Hoek, W.; Meyer, J.-J. C. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems 2*, 4 (1999), 357–401.

[55] Hübner, J. F.; Sichman, J. S.; Boissier, O. Developing organised multiagent systems using the moise+ model: programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering 1*, 3-4 (2007), 370–395.

[56] Jensen, A. S. Implementing lego agents using jason. *arXiv preprint arXiv:1010.0150* (2010).

[57] Junger, D.; Guinelli, J. V.; Pantoja, C. E. An Analysis of Javino Middleware for Robotic Platforms Using Jason and JADE Frameworks. In 10^{th} *Software Agents, Environments and Applications School* (2016).

[58] Kazanavicius, E.; Kazanavicius, V.; Ostaseviciute, L. Agent-based framework for embedded systems development in smart environments. In *Proceedings of International Conference on Information Technologies (IT 2009), Kaunas* (2009).

[59] Keegan, S.; O'Hare, G. M.; O'Grady, M. J. Easishop: Ambient intelligence assists everyday shopping. *Information Sciences 178*, 3 (2008), 588–611.

[60] Kiff, L.; Haigh, K.; Sun, X. Mobility monitoring with the independent lifestyle assistant (ilsa). In *International conference on aging, disability and independence (ICADI)* (2003).

[61] Koda, T.; Elo, S.; Rhodes, B. The coffee robot: An example of ubiquitous computing, 1996.

[62] Lange, D. B.; Oshima, M. Seven good reasons for mobile agents. *Communications of the ACM 42*, 3 (1999), 88–89.

[63] Lazarin, N. M.; Pantoja, C. E. A robotic-agent platform for embedding software agents using raspberry pi and arduino boards. In 9^{th} *Software Agents, Environments and Applications School* (2015).

[64] Leppanen, T.; Liu, M.; Harjula, E.; Ramalingam, A.; Ylioja, J.; Narhi, P.; Riekki, J.; Ojala, T. Mobile agents for integration of internet of things and wireless sensor networks. In *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on* (2013), IEEE, pp. 14–21.

[65] Lim, C.; Anthony, P.; Fan, L. Applying multi-agent system in a context aware. *Borneo Sci 24* (2009), 53–64.

[66] Lippi, M.; Mamei, M.; Mariani, S.; Zambonelli, F. Coordinating distributed speaking objects. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 1949–1960.

[67] Luyten, K.; Van den Bergh, J.; Vandervelpen, C.; Coninx, K. Designing distributed user interfaces for ambient intelligent environments using models and simulations. *Computers & Graphics 30*, 5 (2006), 702–713.

[68] Maciel, C.; de Souza, P. C.; Viterbo, J.; Mendes, F. F.; El Fallah Seghrouchni, A. A multi-agent architecture to support ubiquitous applications in smart environments. In *Agent Technology for Intelligent Mobile Services and Smart Societies: Workshop on Collaborative Agents, Research and Development, CARE 2014, and Workshop on Agents, Virtual Societies and Analytics, AVSA 2014, Held as Part of AAMAS 2014, Paris, France, May 5-9, 2014. Revised Selected Papers*, F. Koch, F. Meneguzzi, and K. Lakkaraju, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 106–116.

[69] Manoel, F. C. P. B.; Nunes, P. d. S. M.; de Jesus, V. S.; Pantoja, C. E.; Viterbo, J. Applying multi-agent systems in prototyping: Programming agents for controlling a smart bathroom model with hardware limitations. *Revista Jr de Iniciação Científica em Ciências Exatas e Engenharia 1*, 16 (2017), 1–8.

[70] Marreiros, G.; Santos, R.; Novais, P.; Machado, J.; Ramos, C.; Neves, J.; Bula-Cruz, J. Argumentation-based decision making in ambient intelligence environments. *Progress in Artificial Intelligence* (2007), 309–322.

[71] Martins, R.; Meneguzzi, F. A smart home model to demand side management. In *Workshop on Collaborative Online Organizations (COOS13)@AAMAS* (2013).

[72] Martins, R.; Meneguzzi, F. A smart home model using jacamo framework. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)* (2014), IEEE.

[73] Masse, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. "O'Reilly Media, Inc.", 2011.

[74] Masthoff, J.; Vasconcelos, W. W.; Aitken, C.; Correa da Silva, F. Agent-based group modelling for ambient intelligence. In *AISB Symposium on Affective Smart Environments, Newcastle, UK* (2007).

[75] Matarić, M. J. *The Robotics Primer*. Mit Press, 2007.

[76] Miorandi, D.; Sicari, S.; De Pellegrini, F.; Chlamtac, I. Internet of things: Vision, applications and research challenges. *Ad hoc networks 10*, 7 (2012), 1497–1516.

[77] Misker, J.; Veenman, C. J.; Rothkrantz, L. J. Groups of collaborating users and agents in ambient intelligent environments. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 3* (2004), IEEE Computer Society, pp. 1320–1321.

[78] Morais, M.; Meneguzzi, F.; Bordini, R.; Amory, A. Distributed fault diagnosis for multiple mobile robots using an agent programming language. In *Advanced Robotics (ICAR), 2015 International Conference on* (2015), pp. 395–400.

[79] Morais, M. G. Integration of a multi-agent system into a robotic framework: A case study of a cooperative fault diagnosis application. Master's thesis, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil, 2015.

[80] Mordenti, A.; Ricci, A.; Santi, D. I. A. Programming robots with an agent-oriented bdi-based control architecture: Explorations using the jaca and webots platforms. *Bologna, Italy, Tech. Rep* (2012).

[81] Moreno, A.; Valls, A.; Viejo, A. *Using JADE-LEAP implement agents in mobile devices*. Universitat Rovira i Virgili. Departament d'Enginyeria Informàtica, 2003.

[82] Mzahm, A. M.; Ahmad, M. S.; Tang, A. Enhancing the internet of things (iot) via the concept of agent of things (aot). *Journal of Network and Innovative Computing 2*, 2014 (2014), 101–110.

[83] Mzahm, A. M.; Ahmad, M. S.; Tang, A. Y.; Ahmad, A. A software-hardware optimizer model for optimized design of things in agents of things. *Journal of Theoretical and Applied Information Technology 94*, 2 (2016), 490.

[84] Negroponte, N. *Being digital*. Alfred A. Knopf, Inc.: New York, 1995.

[85] Ngu, A. H.; Gutierrez, M.; Metsis, V.; Nepal, S.; Sheng, Q. Z. Iot middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal 4*, 1 (2016), 1–20.

[86] Pantoja, C. E.; Jesus, V. S.; Filho, J. V. Aplicando sistemas multi-agentes ubıquos em um modelo de smart home usando o framework jason. In *II Workshop de Pesquisa e Desenvolvimento em Inteligência Artificial, Inteligência Coletiva e Ciência de Dados* (2016).

[87] Pantoja, C. E.; Soares, H. D.; Viterbo, J.; Alexandre, T. S.; Casals, A.; Seghrouchni, A. E. F. A resource management architecture for exposing devices as a service in the internet of things. In *The 31th International Conference on Software Engineering & Knowledge Engineering* (Lisbon, 2019), pp. 215–214.

[88] Pantoja, C. E.; Soares, H. D.; Viterbo, J.; Seghrouchni, A. E. F. An architecture for the development of ambient intelligence systems managed by embedded agents. In *The 30th International Conference on Software Engineering & Knowledge Engineering* (San Franscisco, 2018), pp. 215–214.

[89] Pantoja, C. E.; Stabile, M. F.; Lazarin, N. M.; Sichman, J. S. Argo: An extended jason architecture that facilitates embedded robotic agents programming. In *Engineering Multi-Agent Systems: 4th International Workshop, EMAS 2016*, M. Baldoni, J. P. Müller, I. Nunes, and R. Zalila-Wenkstern, Eds. Springer, 2016, pp. 136–155.

[90] Pantoja, C. E.; Stabile Jr, M. F.; Lazarin, N. M.; Sichman, J. S. Argo: A customized jason architecture for programming embedded robotic agents. *Fourth International Workshop on Engineering Multi-Agent Systems (EMAS 2016)* (2016).

[91] Pantoja, C. E.; Viterbo, J. Prototyping ubiquitous multi-agent systems: A generic domain approach with jason. In *Advances in Practical Applications of Cyber-Physical Multi-Agent Systems: The PAAMS Collection: 15th International Conference, PAAMS 2017, Porto, Portugal, June 21-23, 2017, Proceedings*, Y. Demazeau, P. Davidsson, J. Bajo, and Z. Vale, Eds. Springer International Publishing, 2017, pp. 342–345.

[92] Pantoja, C. E.; Viterbo, J.; Seghrouchni, A. E.-F. From thing to smart thing: Towards an architecture for agent-based ami systems. In *Agents and Multi-agent Systems: Technologies and Applications 2019* (Singapore, 2020), G. Jezic, Y.-H. J. Chen-Burger, M. Kusek, R. Šperka, R. J. Howlett, and L. C. Jain, Eds., Springer Singapore, pp. 57–67.

[93] Pardo-Castellote, G. Omg data-distribution service: Architectural overview. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on* (2003), IEEE, pp. 200–206.

[94] Piette, F.; Seghrouchni, A. E. F.; Taillibert, P. Intelligent agents for preserving resource privacy when deploying ambient intelligence applications. In *Agents (ICA), IEEE International Conference on* (2016), IEEE, pp. 43–50.

[95] Pokahr, A.; Braubach, L.; Lamersdorf, W. Jadex: A bdi reasoning engine. In *Multi-agent programming*. Springer, 2005, pp. 149–174.

[96] Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A. Y. Ros: an open-source robot operating system. In *ICRA workshop on open source software* (2009), vol. 3, Kobe, Japan, p. 5.

[97] Rao, A. S. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world (MAAMAW'96)* (USA, 1996), W. V. de Velde and J. W. Perram, Eds., vol. 1038 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 42–55.

[98] Ricci, A.; Piunti, M.; Viroli, M.; Omicini, A. Environment programming in CArtAgO. In *Multi-Agent Programming: Languages, Tools and Applications*, A. Seghrouchni, J. Dix, M. Dastani, and H. R. Bordini, Eds. Springer US, Boston, MA, 2009, pp. 259–288.

[99] Rivera, D.; Cruz-Piris, L.; Lopez-Civera, G.; de la Hoz, E.; Marsa-Maestre, I. Applying an unified access control for iot-based intelligent agent systems. In *Service-Oriented Computing and Applications (SOCA), 2015 IEEE 8th International Conference on* (2015), IEEE, pp. 247–251.

[100] Rockel, S.; Klimentjew, D.; Zhang, J. A multi-robot platform for mobile robots—a novel evaluation and development approach with multi-agent technology. In *Multisensor Fusion and Integration for Intelligent Systems (MFI), 2012 IEEE Conference on* (2012), IEEE, pp. 470–477.

[101] Rodriguez, M. D.; Favela, J.; Martínez, E. A.; Muñoz, M. A. Location-aware access to hospital information and services. *IEEE Transactions on information technology in biomedicine 8*, 4 (2004), 448–455.

[102] Rodríguez, M. D.; Favela, J.; Preciado, A.; Vizcaíno, A. Agent-based ambient intelligence for healthcare. *Ai Communications 18*, 3 (2005), 201–216.

[103] Russel, S.; Norvig, P. *Inteligência artificial*. Editora Campus, 2004.

[104] Sadri, F. Ambient intelligence for care of the elderly in their homes. In *Proceedings of the 2nd Workshop on Artificial Techniques for Ambient Intelligence (AITAmI)* (2007), pp. 62–67.

[105] Sarwat, A. I.; Sundararajan, A.; Parvez, I. Trends and future directions of research for smart grid iot sensor networks. In *International Symposium on Sensor Networks, Systems and Security* (2017), Springer, pp. 45–61.

[106] Sashima, A.; Izumi, N.; Kurumatani, K. Consorts: A multiagent architecture for service coordination in ubiquitous computing. In *Multi-agent for Mass User Support*. Springer, 2004, pp. 190–216.

[107] Savaglio, C.; Fortino, G.; Zhou, M. Towards interoperable, cognitive and autonomic iot systems: an agent-based approach. In *Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on* (2016), IEEE, pp. 58–63.

[108] Silva, F. S. C. d.; Vasconcelos, W. W. Managing responsive environments with software agents. *Applied Artificial Intelligence 21*, 4-5 (2007), 469–488.

[109] Silva, L.; Endler, M.; Roriz, M. Mr-udp: Yet another reliable user datagram protocol, now for mobile nodes. *Monografias em Ciência da Computação, nr 1200* (2013), 06–13.

[110] Singh, M. P.; Chopra, A. K. The internet of things and multiagent systems: Decentralized intelligence in distributed computing. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 1738–1747.

[111] Soares, H. D.; de Oliveira Guerra, R. P.; de Albuquerque, C. V. N. Ftsp+: A mac timestamp independent flooding time synchronization protocol. In *XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC* (2016), Sociedade Brasileira de Computação, pp. 820–832.

[112] Soriano, Á.; Marín, L.; Valera, Á.; Vallés, M. Multi-agent systems integration in embedded systems with limited resources to perform tasks of coordination and cooperation. In *Proceedings of 10th International Conference on Informatics in Control, Automation and Robotics, ICINCO* (Reykjavik, Iceland, 2013), pp. 140–147.

[113] Stabile Jr., M. F.; Sichman, J. S. Evaluating perception filters in BDI Jason agents. In 4^{th} *Brazilian Conference on Intelligent Systems (BRACIS)* (2015).

[114] Suguri, H.; Kodama, E.; Miyazaki, M.; Kaji, I. Assuring interoperability between heterogeneous multi-agent systems with a gateway agent. In *High Assurance Systems Engineering, 2002. Proceedings. 7th IEEE International Symposium on* (2002), IEEE, pp. 167–170.

[115] Sun, Q.; Yu, W.; Kochurov, N.; Hao, Q.; Hu, F. A multi-agent-based intelligent sensor and actuator network design for smart house and home automation. *Journal of Sensor and Actuator Networks 2*, 3 (2013), 557–588.

[116] Surie, D.; Laguionie, O.; Pederson, T. Wireless sensor networking of everyday objects in a smart home environment. In *Intelligent Sensors, Sensor Networks and Information Processing, 2008. ISSNIP 2008. International Conference on* (2008), IEEE, pp. 189–194.

[117] Tapia, D. I.; Alonso, R. S.; De Paz, J. F.; Corchado, J. M. Introducing a distributed architecture for heterogeneous wireless sensor networks. In *International Work-Conference on Artificial Neural Networks* (2009), Springer, pp. 116–123.

[118] Tapia, D. I.; Rodríguez, S.; Bajo, J.; Corchado, J. M. Fusion@, a soa-based multi-agent architecture. In *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)* (2008), Springer, pp. 99–107.

[119] Titchkosky, L.; Arlitt, M.; Williamson, C. A performance comparison of dynamic web technologies. *ACM SIGMETRICS Performance Evaluation Review 31*, 3 (2003), 2–11.

[120] Vasconcelos, R. O.; e Silva, L. D. N.; Endler, M. Towards efficient group management and communication for large-scale mobile applications. In *2014 IEEE International Conference on Pervasive Computing and Communication Workshops (PERCOM WORKSHOPS)* (2014), IEEE, pp. 551–556.

[121] Villarrubia, G.; De Paz, J. F.; Bajo, J.; Corchado, J. M. Ambient agents: embedded agents for remote control and monitoring using the pangea platform. *Sensors 14*, 8 (2014), 13955–13979.

[122] Viterbo, J. *Decentralized Reasoning in Ambient Intelligence.* Tese de Doutorado, Pontifícia Universidade Católica do Rio de Janeiro (PUC-RJ), 2010.

[123] Viterbo, J.; Mazuel, L.; Charif, Y.; Endler, M.; Sabouret, N.; Breitman, K.; Seghrouchni, A. E. F.; Briot, J. Ambient intelligence: Management of distributed and heterogeneous context knowledge. *CRC Studies in Informatics Series. Chapman & Hall* (2008), 1–44.

[124] Wang, F.-Y. Toward a revolution in transportation operations: Ai for complex systems. *IEEE Intelligent Systems 23*, 6 (2008).

[125] Weber, W.; Rabaey, J.; Aarts, E. *Ambient Intelligence.* Springer, 2005.

[126] Wei, C.; Hindriks, K. V. An agent-based cognitive robot architecture. In *Programming Multi-Agent Systems: 10th International Workshop, ProMAS, Valencia, Spain*, M. Dastani, J. F. Hübner, and B. Logan, Eds. Springer, Berlin, 2013, pp. 54–71.

[127] Weiser, M. The computer for the 21st century. *IEEE pervasive computing 1*, 1 (2002), 19–25.

[128] Wesz, R. Integrating Robot Control into the AgentSpeak(L) Programming Language. In 8^{th} *Software Agents, Environments and Applications School* (2014).

[129] Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M. C.; Regnell, B.; Wesslén, A. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[130] Wooldridge, M. *An Introduction to MultiAgent Systems*. Wiley, 2009.

[131] Wooldridge, M. J. *Reasoning about rational agents*. MIT press, 2000.