UNIVERSITY OF
LEICESTER

THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

Realising Relative Autonomy and Adaptation in Smart Objects Systems

by

Marco Eric Pérez Hernández

Department of Informatics

March 1, 2018

Declaration

I declare that this thesis is the product of my own work, that it has not been submitted before for any degree or examination in any other university, and that all the sources I have used or quoted have been indicated and acknowledged as complete references.
I have previously published some of the contents of this thesis in SMARTCOMP14 [91], FICLOUD15 [92] and FICLOUD16 [93]. I also presented part of the ideas and charts of this project in the BCS Current Leicester Postgraduate Research in Computing 2016.

Marco Eric Pérez Hernández
March 2018,
Leicester

Abstract

The common approach for engineering of applications for the *Internet of Things* (*IoT*) relies heavily on remote resources, particularly in the cloud. As a result, data is collected and functionality is centralised in the cloud platforms leaving devices with only raw data gathering and actuation functions. *IoT* envisions an environment where devices can act as smart objects that are able to make decisions and operate autonomously for the benefit of the human users. Usually, autonomous functions are mixed with automatic functions that only consider the human user point of view.

In this work, we propose an *IoT* application development framework based on goal-directed and role-based smart objects. This framework is composed of a conceptual basis, a software architecture, a middleware architecture and an adaptation method. First, we define the concepts of smart object, its autonomy and the collective of smart objects from a thorough examination of the smart object, its properties and key processes. Then, we develop a set of abstractions and the software architecture for smart objects.

For easing the development effort and making this approach practical, we define a middleware architecture, intended to serve as blueprint for concrete middleware solutions. We also implemented a prototype based on this architecture. Functional components of the architecture enable smart object systems to adapt to volatile situations. We propose a method for adaptation based on the selection of smart objects, services and roles.

Finally, we develop an agent-based model for simulation of *IoT* environments under conditions of heterogeneity, volatility and large quantities of smart objects. We use this model together with a case study and a qualitative comparison of existing solutions to evaluate our framework. Our results show that the proposed approach is a feasible and scalable alternative for *IoT* application development based on smart objects that incorporates the concept of relative autonomy, in this context, and the adaptation at individual and collective level.

Acknowledgements

There are several persons I would like to thank.

First of all, I would like to thank Dr. Stephan Reiff-Marganiec for his supervision along my PhD work. His advice, motivation and support has been fundamental for the development of this thesis. I appreciate his continuous guidance and his experience for providing always valuable contributions, advice and moral support.

I would also like to thank Professor Reiko Heckel and Dr. Rami Bahsoon for their valuable suggestions and opinions that were very useful for the final version of this work. Likewise, I would like to thank Dr. Emilio Tuosto and Dr. Fer-Jan de Vries for their relevant opinions along the different stages of this project. I also want to thank Ludovic Clarissou from Télécom Saint-Étienne, for help me in the validation of the *em4so* middleware prototype.

Special thanks also to several people of the department that directly or indirectly support me along this journey. Thanks to Dr. Gilbert Laycock, for his opinions and suggestions, particularly during the evaluation. For this project, I have learned and gathered ideas from my teaching assistant activities, particularly working with Dr. Stephan Reiff-Marganiec, Dr. Artur Boronat, Dr. Yi Hong and Professor Rick Thomas. Many thanks to all of them.

I want to thank the Department of Informatics of the University of Leicester for funding my research studies. Besides the persons already mentioned, many thanks to Professor Thomas Erlebach, Professor Alexander Kurz and Professor Effie Law.

Thanks to my colleagues PhD Students, together we listened to each other, gave mutual support. Thanks specially to Laith, Hao, Othman and Badr.

Last but not least, I want to thank my wife, parents, sister, family and friends. This thesis is also yours. Thanks to Nidia for her love, support, patience and strength during the ups and downs of this journey. Thanks for all her encouragement and for being part of this. Thanks to Marco Antonio, Martha and Bibiana, their love, motivation and advice give me always that extra boost needed. Thanks to my friends in UK, Spain, US and Colombia, particularly, to my friend Oscar for his frank and timely advice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past few years *Internet of Things* (*IoT*) has become an active research field where multiple areas from computer science and other disciplines converge. *IoT* is envisioned as a paradigm based on the idea of regular objects that are interconnected through the internet, combining cyber physical functionality and data gathering in order to make human users life easier. *IoT* scenarios are characterised, among others, by the cyber physical operation, heterogeneity, volatility and human user orientation. In a first stage, the research community has been working vigorously to tackle the open challenges of device connectivity, interoperability, data gathering and processing, among others. A popular *IoT* application development approach has emerged naturally, incorporating advances in tackling these challenges and particularly, the inherent heterogeneity.

Considering the constrained computing capabilities of the *IoT* devices, most of the software engineering approaches have been directed towards concentrating the most of the functionality in the cloud platforms. This is an effective way to tackle heterogeneity as devices are paired to the lower capacity, enabling applications to deal with every device uniformly, regardless of their individual differences. As a result the *IoT* software architecture of these applications is realised as having a "physical layer" where the individual devices are conceived as kind of "cyber physical peripherals" that only gather and distribute raw data and actuate on the environment upon request, regardless of each individual device's characteristics. However, considering the volatility of the scenarios, the human user demand for trustworthy applications and the increasing power of the *IoT* devices. The *IoT* paradigm is also expected to enable the development of applications based on smart objects (*SOs*), that is, objects able to make decisions, operate autonomously and cooperate with each other in order to carry out individual and collective tasks.

The *IoT* based on smart objects is still under construction and while multiple challenges exist, this approach is emerging as an alternative to the established one. In this work we address and contribute towards filling some of these gaps. One first challenge is at conceptual level: there is no agreement on what a smart object is, its properties and, more importantly, what it means to be autonomous for a smart object. Therefore, our first contribution is to thoroughly explore and analyse the smart object characteristics and based on these, propose definitions for smart object, collective of smart objects and smart object's autonomy, as key concepts for a software engineering based on smart objects. This analysis subsumes existing literature, particularly in the areas of agent and ubiquitous computing. As a result, we build the concept of autonomy, not only relative to the human user perspective, but also from the point of view of exogenous platforms and other smart objects. In order to bring relevant elements together, we also propose an analysis tool based on the identification of key areas, fundamental processes and operation planes.

Another challenge at the smart object's software level, is how to incorporate the defined concepts as part of the software development effort and identify the components, relationships and functionalities that have to be covered in order to enable *IoT* devices to become smart objects, considering the aforementioned typical *IoT* scenario conditions —E.g. heterogeneity, volatility, etc.—. A reference *IoT* software architecture must provide blueprints and abstractions for the development of *IoT* applications enabling sharing of expertise, among different applications, in the shape of behavioural and structural solutions. To tackle this challenge, we propose a set of abstractions that together build the software architecture of a smart object and provide the building blocks for an alternative *IoT* application development approach. These abstractions are based on those from the agent, service and role-based computing paradigms. The set of abstractions and architecture brings a novel approach for smart object software development that endows a relative definition of autonomy, dynamics and flexibility to the *IoT* systems.

For a software architecture to be practical, the development effort must be reduced as much as possible to the specific application logic required, instead of requiring engineering of software from the scratch to every application. For this purpose and since the early stages of the *IoT* paradigm, middleware has been identified as a key enabler of the common requirements of *IoT* applications including support to heterogeneity and

volatility. There are several middleware solutions designed under the established development approach of the "physical layer", there are also few solutions aimed at smart objects development. However, the multiple views of autonomy are not considered in any of these. Therefore, another contribution of this work is the definition of a middleware architecture that enables the engineering of concrete middleware solutions that support common operations of smart objects inline with the proposed software architecture and application development approach.

For the middleware architecture, since we have to consider multiple views of the autonomy, the traditional operational approach of agent platforms as controlled environments and imposing particular models of integration with the internet, does not fulfil the requirements of smart object's autonomy. We investigate on how to remove constraints of an enclosed agent platforms by enabling smart objects based on web agents. In addition, since smart objects do not have the endless resources a cloud platform can provide, a mechanism for cooperation and social interaction between smart objects is fundamental, this way, they overcome individual limitations and engage in achieving common goals.

Considering the characteristic volatility of *IoT* scenarios, adaptation is a fundamental common feature of the *IoT* applications and then ideally offered by middleware solutions. The challenge is in how to enable smart objects and collectives of smart objects to adapt to changing conditions at individual and system level. We address this challenge by defining runtime selection methods for the most suitable smart objects, services and roles at each given situation.

Finally, with the lack of methods that incorporate the different variables that take part in smart objects systems, the evaluation of these systems is still an open challenge. Our contribution in this regard is the definition of an agent-based model that enables to test medium-scale collectives of smart objects under conditions of variability and heterogeneity.

As a whole, this work offers a framework for *IoT* application development based on smart objects that are goal-oriented and role-based. As opposed to other smart object approaches, this proposal considers multiple views of autonomy and from that, it defines the abstractions, software architecture, middleware architecture and adaptation methods required to provide a concrete toolbox for application development.

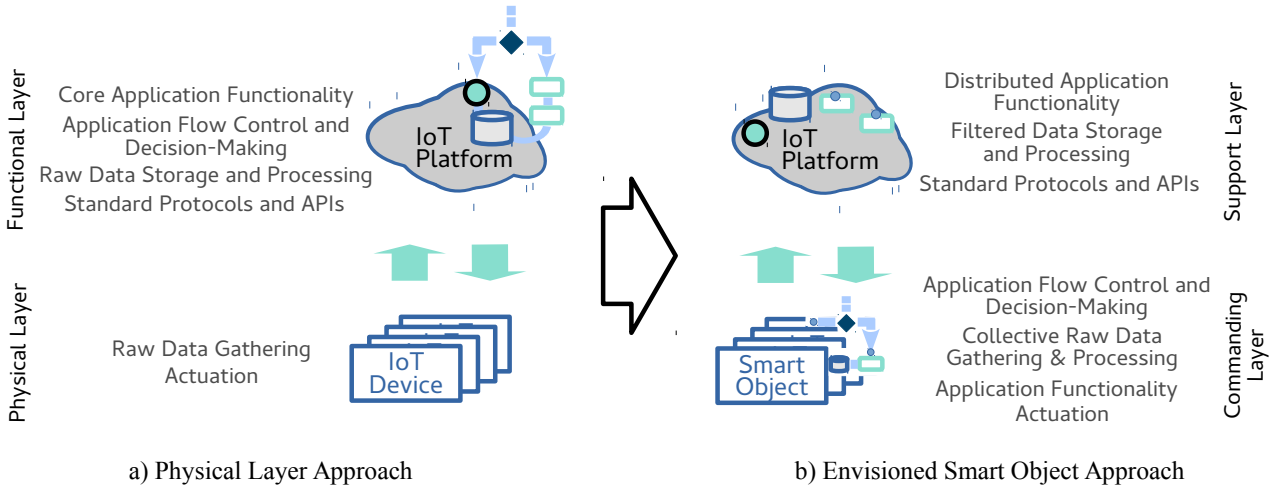


Figure 1.1: *IoT* Application Development Approaches

This work has been partially published in SMARTCOMP14[91], FICLOUD15[92] and FICLOUD16[93].

The remainder of this chapter will explore further the research challenges, contributions and thesis statement. Besides, it will provide an overview and organisation of the whole document.

1.1 Research Problem and Challenges

Most of the popular approaches for *IoT* application development conceive the *IoT* devices as data sources that feed web applications, mainly deployed in cloud or network platforms. From an architectural perspective, the application logic, flow and decision-making is controlled and centralised from within the platform in a model similar to the presented in figure 1.1 a). Raw data gathered from the devices is consistently stored in remote repositories. These models present various advantages, for example: a) these platforms make intensive use of cloud infrastructure for enhancing device capabilities and scope, enabling more complex applications; and b) these offer a portfolio of runtime services covering device management, data analytics and enterprise integration, among others. However, the following key issues have been identified with this model:

- *Functionality Concentration*

  The minimal functionality required by the user has to go through the remote platform and then come back to the device with a response to the user, even if both input and outcome are only related to the device and the device has the

power to carry out the functionality locally. As a result, the advantage of the device's unique cyber physical characteristics —E.g. location— is diminished by the control and decision-making exercised remotely.

- *Lack of device autonomy*
  Since the control of the application flow and the core functionality is kept out of the device, it becomes useless in situations of unstable or failing network connection. Part of the vision of the *IoT* involves autonomy of devices, however this is usually perceived from the perspective of the user. Hence, the devices seem autonomous because they can carry out some functionalities without user intervention but the operation is controlled by a remote application. The moment, the connection is lost, the "autonomy" and complete functionality of the device is also lost, indicating that this is not a truly autonomous device.

- *Data Accumulation*
  All gathered data is stored in the remote platform even if it is not required by the application. Platforms can potentially have access to many users data that could be shared without the user being informed. Having user's data concentrated makes it more exposed to security and privacy threads. This is more critical in personal and home applications, where devices are constantly gathering data even when the user is not aware of.

With these issues in mind and being aware that, even though we are entering a post Moore's law era [114], where the increase in computing power for devices will not be as fast as it used to be, there are still perspectives of hardware platforms for *IoT* devices to continue becoming more powerful —E.g. the Intel® Joule™ Platform[1] —. We join the view that a different approach for *IoT* application development is possible where the *IoT* devices have more responsibilities. In that vision, the *IoT* devices become smart objects that are autonomous and able to control the applications using the remote resources as support of their operation. Such a vision is presented in figure 1.1 b).

This view of *IoT* applications, where the control is kept in the devices is particularly suitable for scenarios with two broad requirements. First, the privacy is important, users are not comfortable uploading data about their everyday activities and storing it out of their control; and second, the activities done by the smart object are simple as it is not worthwhile to couple the device operation to a remote platform while constraining device's autonomy. The remote platforms are indeed used, but as support of the smart

[1] https://software.intel.com/en-us/node/721455

object and not as the primary source of its functionality. These scenarios are found in the smart home and smart personal space domains; we describe one leading scenario as part of out evaluation in chapter 7. Many challenges are relevant to this alternative approach, we are going to focus on the following:

R1 In the research community and industry there are several *IoT* solutions claiming to be "autonomous" because they provide some automatic functions or use some of the ideas of software agents, however there is not a common understanding of what autonomy implies in the context of smart objects. This concept needs to be revisited and defined in this context, to build a conceptual basis that enables definition of the basic common requirements in regards processes and functionality to be satisfied by the smart object's software to ensure this characteristic. Key issues to address include:

- What is smart object's autonomy and how does it differ from other more abstract concepts such as agent autonomy?
- What are the elements and processes of the smart object that shape its autonomy?
- How is the smart object's autonomy constrained and how can this be avoided from a software engineering perspective?

R2 Define an *IoT* software architecture based on smart objects that enables them to be autonomous and keep control of the *IoT* applications. This approach must include abstractions for managing the typical heterogeneity of smart objects and exploiting local resources while enabling the use of the remote resources as support and the cooperation with other smart objects to enhance own capabilities. Key issues to address include:

- What are the key requirements for a software architecture for smart objects?
- How can functionalities of the smart objects be defined regardless of their differences?
- Which components should the smart object's software include to be able to operate autonomously but also cooperate with others?

R3 Specify a middleware architecture that enables the provision of smart object's common functionalities as runtime services and programming libraries. Rather than a single middleware solution that constraints the development to concrete supported technologies or programming platforms, this middleware architecture should serve as reference, offering abstract components, relationships and the

means for their extension according to the developer needs. Key issues to address include:

- Which common functionalities should be offered by a middleware aimed to enable software engineering for autonomous smart objects?
- How can smart object middleware enable extension of smart object's functionalities with reduced effort while preserving autonomy?
- How can middleware enable cooperation between smart objects and taking advantage of their collective operation?

R4 Since *IoT* environments are volatile, it is clear that one of the key features of any middleware is adaptability of the *IoT* systems to different conditions. Aligned with the defined concept of autonomy, a mechanism for adaptation of smart objects must enable to take advantage of other smart objects and remote services when these are available, without constraining its control of applications and ability to operate without them. Key issues to address include:

- How can a system of smart objects achieve cooperative goals without depending in particular smart objects?
- How can a system of smart objects cope with changes in its components i.e. the smart objects?
- How can smart objects adjust their own structure and behaviour to the context?

R5 A derived challenge from the previously defined, is the definition of an approach for evaluation and testing of *IoT* systems in conditions of heterogeneity and volatility considering large quantities of devices. Nowadays, existing testing and evaluation techniques are insufficient as they offer only partial coverage to the aspects mentioned or do not allow to test software and middleware architectures with the expected number of participating devices.

1.2 Thesis Statement

This thesis analyses the smart object's autonomy and proposes a set of concepts and an *IoT* software architecture on top of it. The software architecture is based on goal-oriented role-based smart objects that cooperate within a collective. A middleware architecture is the enabler to provide common functionalities. Among these, we define

a method for collective and individual adaptation of smart object systems based on the selection of smart objects, services and roles.

1.3 Research Scope and Contributions

The main research contributions of this thesis are:

C1 For tackling R1, we propose a definition of the concepts of smart object autonomy and collective of smart objects and a thorough examination of the *SO* as an autonomous system, its properties and processes. Together these concepts and the *SO* characteristics detailed augment the existing key literature —E.g. [62, 69, 32]— by providing the basis for a *Smart Object*-based *IoT* application development with attention to the *SO* autonomy and the role of an *SO* within a collective (Chapter 3).

C2 For addressing R2, we present a novel architecture for *IoT* applications based on goal-directed and role-based smart objects (*RbSOs*). This architecture enables the development of *SO* systems that hold the control of key processes, application logic and data, being able to work at different levels of autonomy, not only from human user perspective (Chapter 4).

C3 For tackling R3, we created a middleware architecture covering common functionalities for the development of *RbSOs* software. This architecture works as a template for the development of specific middleware solutions that offer runtime services on-object, covering key *SO* operations. It also incorporates the definition of an *SO* protocol that enable coordination and cooperation within a collective of smart objects and following the *RbSOs* approach (Chapter 5).

C4 We tackle R4 by defining a decentralised utility-based method for individual and collective adaptation of smart objects based on the *RbSOs* approach and the *em4so* middleware architecture. This covers a strategy based on the selection of *SOs* to cooperate with, the selection of services to carry out a plan and the selection of roles to offer within a collective (Chapter 6).

C5 For addressing R5, we propose an agent-based model for the evaluation of *IoT* systems based on smart object and incorporating heterogeneity, instability and large number of nodes. This model enables to simulate installation of middleware and software of top of the smart objects and monitor the behaviour of the *SOs* and the collective during operation (Chapter 7).

1.4 Thesis Overview and Summary

This chapter has presented the rationale behind this work and provided an overview of the contributions. We showed the key challenges for a smart object-based *IoT* application development approach that considers multiple views of autonomy. We scoped our work through a thesis statement focused in the building of autonomy foundations in the context of smart objects. From there, we define a set of abstractions and software architecture for the smart objects and propose of a middleware architecture with special emphasis in offering relative autonomy and adaptation at individual and collective level.

The remainder of this thesis is organised as follows:

- In chapter 2, we provide details of the background of this work. Starting from an overview of the *Internet of Things* field, we continue in more detail presenting the key paradigms and concepts that are used for building *Smart Object*-based *IoT* applications. We also summarise the main related work in the area of smart objects software development and middleware.

- In chapter 3, we elaborate the concept of smart object's autonomy from the organised examination of the properties of the smart object. We also propose definitions for smart object and collective of smart objects. Besides, we present the analysis tool we developed in order to carry out the structured analysis of the smart object as an autonomous system.

- In chapter 4, we introduce the overall approach for engineering of software for smart objects. We present the main abstractions and how these are organised around a software architecture.

- In chapter 5, we identify the common functionality that a middleware based on the previously defined approach has to cover in order to ease the development effort. We organise these functionalities as a middleware architecture, making concrete the abstractions presented in chapter 4. We also defined a smart object protocol that enable the cooperation and coordination based on the functional components of the architecture. Finally, we present details of the implementation of an actual middleware prototype, based in the proposed architecture.

- In chapter 6, we describe the selection methods that enable adaptation of smart object as a key common functionality offered by the middleware architecture. We

define a set of utility functions that incorporate the main criteria to consider in selecting smart objects, services and roles. We explain how these selections methods work together as enablers of adaptation at individual and collective level.

- In chapter 7, we detail our evaluation approach that includes the proposal of an agent-based model of smart object systems. We also present the results of case study implemented based in a real testbed from the prototype introduced in chapter 5. We measure performance of middleware functionalities at individual level in a real setting and at collective level in a simulated environment based on the agent-based model. Finally, we compare our middleware architecture with that of the main solutions available and discuss the results obtained.

- In chapter 8, we present the summary of this thesis and reflect on the main conclusions and contributions. Finally, we outline a set of potential work streams for future research efforts.

Chapter 2

Research Background and Related Work

2.1 Overview of the Internet of Things

The Internet of Things (*IoT*) is the result of efforts in different computer science and other science, engineering and even arts disciplines. Although it is clear that it represents a revolution in everyday life, the foundations are still under construction and agreement within the research community and the industry. Depending on the area from which the *IoT* concept is approached it has different elements and focus.

One of the first *IoT* definitions, by the CERP-*IoT* [105], envisioned *IoT* as a network infrastructure with virtual things integrated in an information network. The most popular view of *IoT* is as a paradigm, where the existence of internet-connected digital entities, representing the physical things, enables the realisation of multiple use cases. Uckelmann et al. [109] stress that, by means of these representations, the right information is available and accessible at the right quantity, condition, time, place and price. They distinguish *IoT* from other approaches such as ubiquitous/pervasive computing, embedded devices, the intranet/extranet of things, although recognise that it combines elements of these. On the other side, Vermesan et al. [112] aim attention at the pervasiveness, interaction and cooperation abilities of the things for the creation of new services/applications.

Atzori et al. [8] initially surveyed and presented an *IoT* conceptualisation based on the convergence of the things, internet and semantic views; where the spotlight was on the augmentation of objects, the inter-connection and the importance of semantic

technologies for the use of generated information. More recently, the same authors [10] propose a definition coming from an analysis of the *IoT* evolution since its origins. They see *IoT* as a "conceptual framework" where heterogeneity, interconnection and shared information of augmented objects, at global scale, are key for the design of applications enabling involvement of people and virtual objects.

The IEEE *IoT* Initiative [80] envisions *IoT* as a network and provides two definitions depending on the complexity of the scenario. They define a "small environment *IoT*" as "a network that connects uniquely identifiable Things to the internet", this is centred in the collection of things information and change of their status without restrictions of place, time and user. Their "large environment *IoT*" is "a self-configuring, adaptive, complex network that interconnects Things to internet through standard communication protocols...". Then, they elaborate the concept from a wide set of features including sensing, actuating, programmability, service-offer and intelligent interfaces, among others.

Another term commonly used to refer to the same or a very similar concept is Cyber-physical system (CPS). A CPS is regarded as "the integration of computation with physical processes" [72]. From our review, it is more popular within engineering fields —i.e. might involve electronic and mechanical components and not only software— and it clearly emphasises the system nature of the concept. Authors of [80], also indicates that CPS might work at intranet as well and not only at internet scale.

It is evident that reaching agreement on a definition is challenging and it is out of scope of this work. For this work, we will adopt the view of *IoT* as an umbrella paradigm where all the other approaches and pre-existing fields have room. We summarise the set of key features identified along the reviewed contributions, these are presented in figure 2.1. *IoT* conceives an ubiquitous, global scale internet which is the base for communication and interconnection of heterogeneous augmented things. It enables the interaction between things and people by providing effective and secure information and services that pervade and gather from the physical environment through sensing and actuating capabilities. The *IoT* enables programming of different behaviours and is able to self-configure.

Figure 2.1 also shows, at the top, some of the key fields that have been integrated in the *IoT*. At the bottom, some of the branches derived from *IoT*: Web of Things (WoT)

[28], Social-*IoT* (S-*IoT*) [9], Cognitive *IoT* (C*IoT*)[119] and Smart Object-based *IoT* (*SO*b-*IoT*) [69]. Within the remaining sections of this chapter we will mention some of these branches giving more attention to WoT and *SO*b-*IoT* which is where this work is situated.

2.1.1 *IoT* Realisation Models

In 2015, The Internet Architecture Board (IAB)[1] identified four communication patterns commonly used in *IoT* [108]. The communication approach depends on the characteristics and, particularly, the communication features of the devices. Likewise, these characteristics impose constraints for the software architecture of the solutions. E.g. when the solution requires support to devices unable to process data, then the software architecture for the solution must include functionality for data collection off-device. Figure 2.2 presents an extended version of the communication patterns, which are described below. We call it realisation models as, instead of addressing connection and interoperability considerations as IAB did, we take a software architecture-oriented perspective.

Device-To-Device (D2D)

As presented in figure 2.2a, the idea is that devices connect directly to each other. This connection is usually based in proximity or even visual field, rather than a truly internet connection. Technologies for achieving this model include communication protocols such as Bluetooth and more recently Zigbee and Z-wave; but we can also consider identification mechanisms such as RFID.

The architecture of the solution might follow a typical Client/Server model where the server is a powerful device. The server device carries out the main functionality, data processing and usually interfaces with the user. The client is only considered as a data source or peripheral, instead of adding real functionality to the solution, it depends absolutely on the server. It might be the case, that devices involved have enough power to work in a distributed peer-to-peer communication and processing model. In this scenario, each device carries out part of the functionality of the *IoT* solution.

[1] https://www.iab.org/

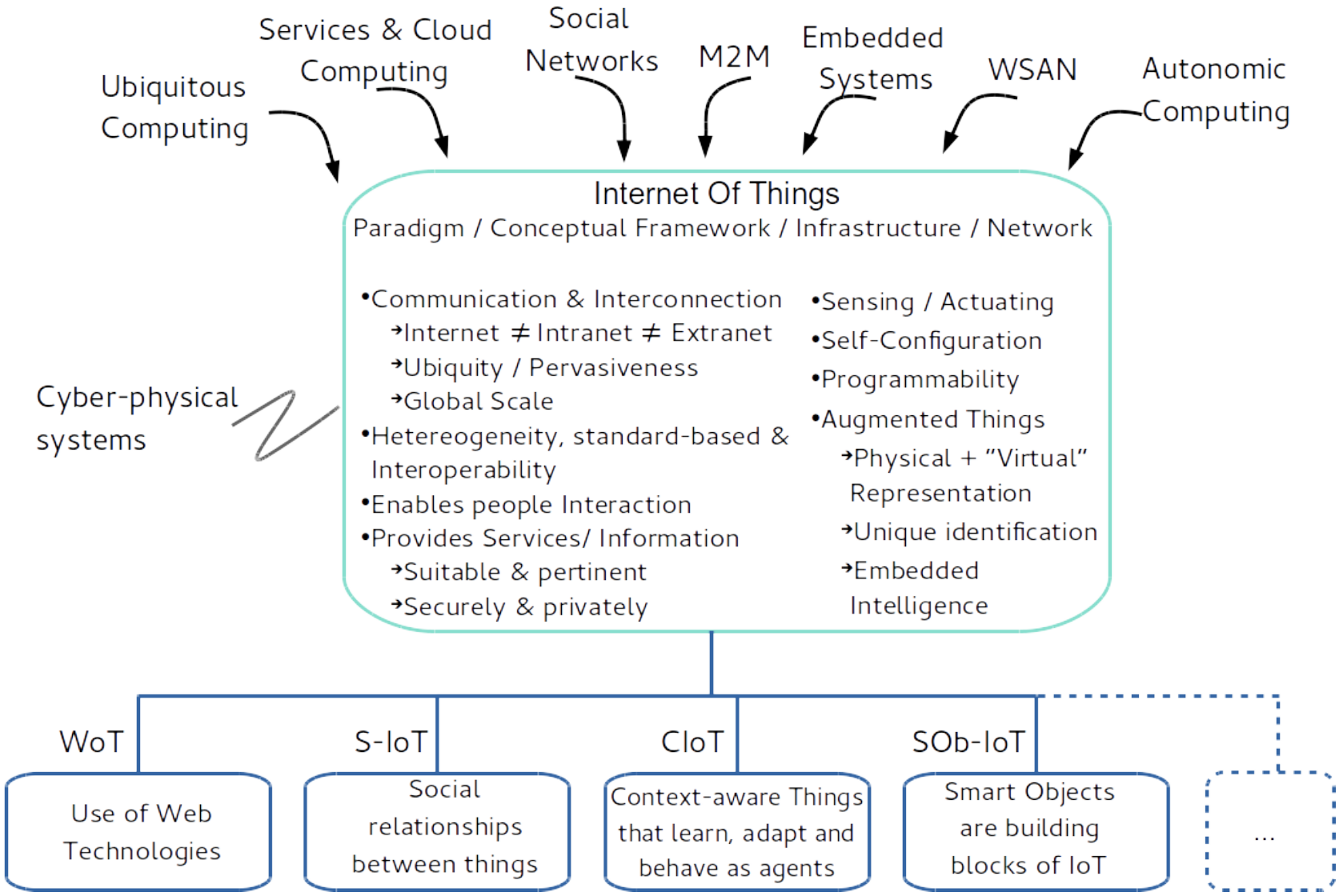


Figure 2.1: *IoT* Concept, characteristics and main branches in the literature.

Device-To-Cloud (D2C)

The devices are capable of connecting directly to the internet as shown in 2.2b. The connection is over IP using available transports (E.g. TCP, UDP) and web, publish/-subscribe message queues or ad hoc constrained protocols such as COAP[2] for the application layer. The devices are generally connected to cloud platforms that provide access to multiple services over a virtualised hardware infrastructure. We observe two software architecture approaches for the *IoT* solutions:

- ***"Dumb Data Feeders"***
  The most typical architecture is based on placing the core functionality in the Cloud Service Provider (CSP). Therefore, devices systematically send raw data to the CSP, which is in charge of collecting, storing and processing it through multiple applications or services. The CSP has the control of the data and the application logic flows. This approach has several advantages such as:

  - It provides a normalised interface to heterogeneous devices.
  - Since functionality is in the cloud, it offers wide support even the most constrained devices that can be used as data sources.
  - Application is centralised in the CSP making it ease to maintain and enhance.
  - If one device is not available, users can still use already collected data.

  However, some of the drawbacks include:

  - The whole raw data, sometimes even unnecessary for the application is collected and stored in the CSP. Such concentration of data raises privacy issues such as how, for how long and who might have access to and use it.
  - The more powerful devices are sub-utilised in order to have normalised interfaces.
  - It is up to the device manufacturer to couple it to a specific cloud service provider which causes vendor lock-in.
  - There is no true autonomy nor "intelligence" in the devices in fact, assuming that they all are equally constrained to a minimum of resources, reduces the computing value to the mere connection, which hampers the potential of the *IoT*.

[2]Constrained Application Protocol

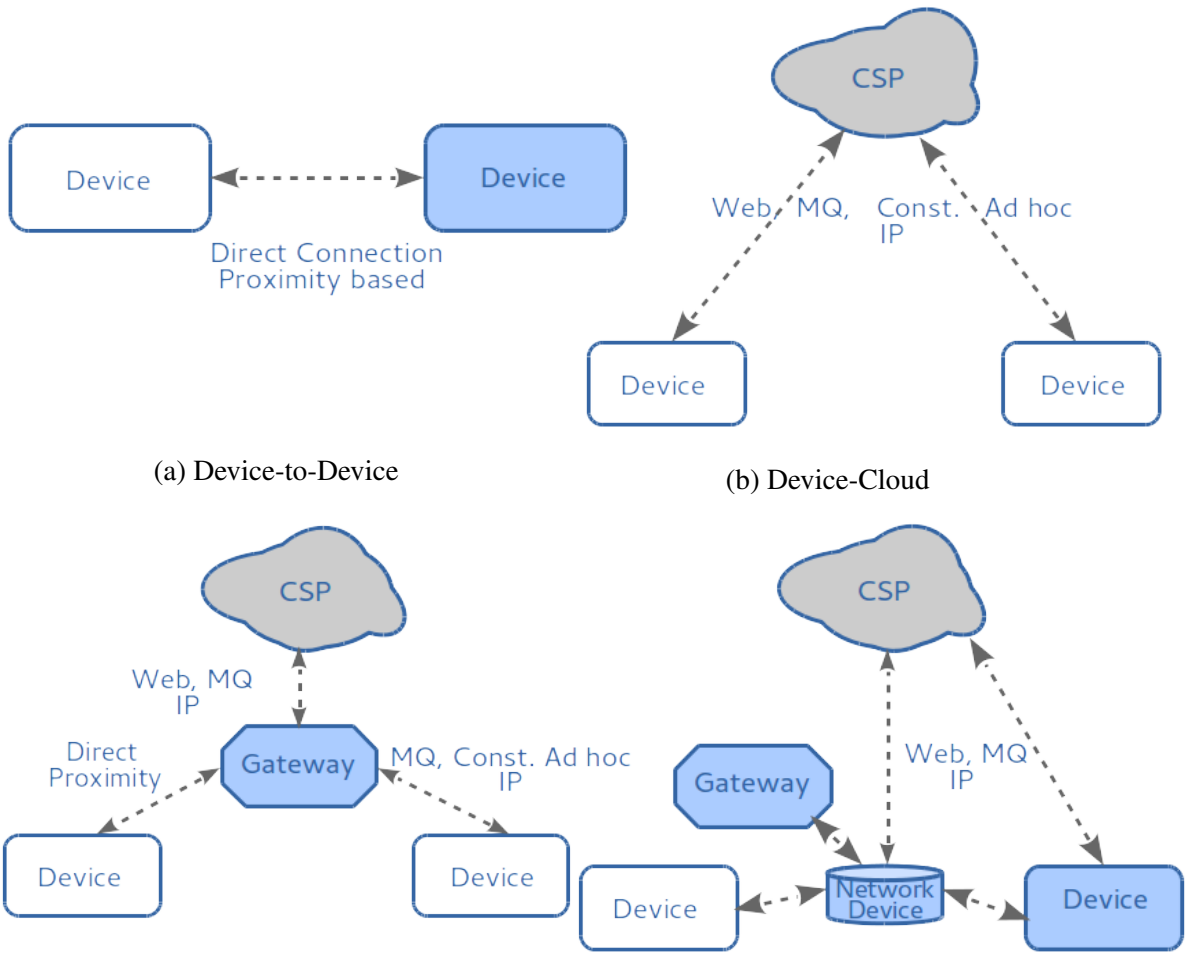
CSP might offer web applications that enable users to change behaviour of the devices or the whole solution. The provider might also offer APIs (E.g. libraries or REST-based) to enable access of the gathered data from ad hoc applications and systems. This approach is identified by IAB as a separate communication pattern called "Back-End Data-Sharing"(BEDS) [108]. The BEDS pattern introduces the aggregation of multiple data providers so users can have integrated views —E.g. Service mashups— that are possible in any case having the data in the cloud. In this case the CSP behaves as a data repository that is queried by multiple applications, each one having its own logic.

- ***"Embedded Intelligence"***
  An alternative, less exploited, approach for architecture is based on keeping part of the functionality of the solution in each device. In this case, the cloud is only used for supporting and backing device functionality on-demand rather than giving the CSP the whole control and responsibility of it. The advantages of this approach are:
  - CPS are only used for specific tasks and so only a portion of the data is sent to them. Even multiple CSP might be used, then reducing the data concentration on a single platform or provider.
  - The CSP vendor lock-in is avoided as devices can work with multiple platforms and providers.
  - *IoT* solution is more resilient as these are able to operate even if the internet connection is unavailable.

  On the other side, the disadvantages are:
  - Different devices require different software components and hence development of maintenance of the solution becomes complex.
  - Some data or functionality might be only available if a particular device is connected.

Device-Gateway-Cloud (DGC)

This approach is presented in figure 2.2c, the main purpose is to enable internet connectivity to devices that do not have it directly. It has the potential to be used not only to fit this purpose but also to collect, aggregate, pre-process the data gathered from constrained devices and to carry out part of functionality of the *IoT* solution. The gateway



(a) Device-to-Device

(b) Device-Cloud

(c) Device-Gateway-Cloud

(d) Edge

Figure 2.2: *IoT* Realisation Models: Expanded from IAB Communication Patterns [108]

is generally a mobile phone, tablet, laptop, PC, a dedicated computer or ad hoc hub.

The architecture of the *IoT* solution varies according to the role given to the gateway. It might only send the data to the CSP as in a Device-Cloud solution or it might provide additional functionality. It has the advantage that enables support to more constrained devices than the Device-Cloud model. However, it adds an additional device to manage and control to the solution that, if used just to send raw data to the CSP, raises the question about which is better: to design a solution giving support only to devices with internet connectivity or giving support to any device, by adding a new one that enable them to connect to internet.

Device-Fog/Edge (D2F/E)

Finally, the Device-Fog/Edge realisation model highlights the localisation of the hardware resources and the nature of the virtualisation layer supporting the operation of the devices in an *IoT* solution. In the Cloud model the virtualised hardware platform comes from dedicated data centres, the Fog extends this power by including devices and network equipment that provide a geographically distributed hardware infrastructure over which application and services are offered in proximity to where these are requested [1, 110, 17].

The Fog/Edge computing is a paradigm even newer than *IoT*. The constraints and potentials are, to date, being determined, with platforms still in the early stages. So far, main changes are from the virtualisation layer, keeping operating system and application layers as they work normally in cloud environments but enabling use of distributed physical resources that include network equipment e.g. routers. Little can be said about software architectures of *IoT* solutions using this realisation model as these are still to appear. However, since the Fog model is based on a platform that manages the Fog resources, it is reasonable to expect a model where highly constrained devices feed a central platform with raw data with the underlying virtualisation manager splitting the load across the available edge and cloud resources.

In contrast to the D2D approach, that enables direct connection between devices via proximity protocols, the D2F/E uses distributed computers and network equipment — e.g. switches or routers— for enabling communication as well as data processing, storage and other functions to explore. The network equipment, do not only meet a network purpose but also an application-level one. Scenarios covered by the D2D approach are

of lower scale than the ones covered by the D2F/E. Examples of D2D include applications for controlling light bulbs, doors and blinds in smart home or industrial scenarios, where the controlled device are paired to controlling device without any extra network equipment but limited to the physical range of the protocol. On the other side, one example of D2F/E in a smart industry can involve devices in multiple factories' shop floors that are connected to internet via routers. These devices are available to applications whose functionalities, processes or services are distributed, deployed and executed in the routers or devices with computing power in each shop floor, where it best suit the application's user.

2.2 Web of Things

Since heterogeneity is one of the key characteristics of *IoT* and a single language or communication protocol for *IoT* solutions is unrealistic, engineering of *IoT* solutions involves many protocols, programming languages, operating systems, architectures and, in general, communication and software technologies. For a solution, it is difficult to ensure support to even the most of the key protocols and technologies within the *IoT* spectrum. As a result, *IoT* solutions become silos only able to communicate with others coming from the same manufacturer or built over the same technologies. These are not able to connect to internet but instead work in a kind of intranet in a model similar to the Device-to-Device realisation model. From the software engineering perspective, the integration efforts within a single solution or with others become extensive.

Thinking on these issues and others, the research community has promoted the use of web notions and standards in application development and communication for *IoT* solutions in what is called the Web of Things (WoT). Guinard [43] states that the ultimate goal of WoT is to maximise the use of tools and techniques coming from the web arena to apply them in the *IoT* scenarios. For doing so, web protocols are the solution for inter-operability of multiple solutions and provide truly wide internet connectivity. Therefore, WoT aims to concentrate efforts around the application level of the protocol stack (OSI model layers) and abstract from the complexity of lower layers (E.g. transport, network, etc.). Referenced from [43], key benefits of WoT over *IoT* include: Use of open standards, easier deployment, maintenance and integration; loose coupling between elements and the existence of widely used security and privacy mechanisms. On the other side, two of the major drawbacks identified so far are: the risk of enabling web

wide access to the *IoT* devices, even with security mechanism in place; and the "general purpose" nature of the web standards might not be optimal in low-resource devices.

The service-oriented computing (SOC) paradigm is part of the essence of modern web applications and so it is for WoT solutions. The approach for developing WoT applications is that "things" expose their functionality —E.g. based on physical sensors— as web services. Wu [120] describes web services as "a kind of application, based on a web environment, that is adaptive, self-describing and modular and has good interoperability". Web services are published by providers and then queried, found and bound/invoked by consumers: either applications or other services. This approach is then advantageous as web services offered by "things" can be reused in multiple applications with minimal effort, regardless of the underling hardware and low-level network connectivity. In WoT, "things" have web service hosting capabilities and they are able to consume others' web services. There a two well known architectural styles for building web services: RPC and RESTful, we briefly described them based on [97].

The RESTful style is based on the identification of resources, with an URI (Unified Resource Identifiers), over which a set of operations can be performed. What is defined as a resource depends on every application, the operations are given by the HTTP protocol —i.e. GET, POST, PUT and DELETE.—. Hence, RESTful web services are explicit in that the URI indicates what operation is being carried out over the resource and there is not additional processing required on client or server side to unwrap the request and response. Conversely, RPC Web Services use data wrappers for transmitting both request and response. It means that both client and provider have to unwrap what they have received in order to determine the contents. The wrap is usually a HTTP request and the contents is the meaningful information in a document format E.g. XML, JSON, etc. The content varies according to the type of information being transmitted and the operation requested.

Although the RESTful style is more popular along WoT applications, both of them are valid for different scenarios and are called to coexist according to particular requirements [125]. The abstraction level in which this thesis is developed goes beyond a particular web service style. There is no restriction to apply one or the other.

In WoT, web services usually gather data from or trigger actions on the physical environment. Main differentiating factors among WoT web services come from the phys-

ical properties of the provider —E.g. location— and the devices —namely sensors and actuators— linked to it. It is not the case that any service can be deployed in any server, these services are more coupled to the provider than traditional web services. Implementation of these services depends on the hardware platforms and the specific devices which are part of the "thing" and that vary from one manufacturer to other. However, the architecture and operation of these services is the same that in traditional web services.

Composition of various web services brings more complex and powerful applications. Traditional service composition methods are also valid in the WoT domain. Considering the method, there are two basic types of service compositions: orchestration and choreography [88]. In orchestration a business process is created from a flow of interacting web services while keeping the control from a single point. On the other side, a choreography does not follow a single process but the sequence of messages among different providers and consumers. Considering the composition time, it could be static –before runtime— and dynamic —during runtime—. Considering the automation level, there are manual techniques —requires an human design and binds the composition—, automated —the composition is generated from a requirement and using mainly semantics and AI— and semi-automated —a mix where the human designer is supported with some automatic processes— [100].

The lifecyle of a service composition includes four generic phases [100]. First, the definition of the composition requirements by the designer, including control flow and QoS. These requirements are the components of the service abstract process model. Second, the selection of the particular services that make concrete the model. Since there might be multiple services fulfilling the functional requirements, these services are selected based on the non-functional properties. Third, the deployment of the composite service, where the bindings to each individual service are made. And finally, the execution or invocation of the composite service.

2.3 Smart Object-based *IoT*

One of the approaches for building the *IoT* vision is based on the concept of Smart Object (*SO*). In this approach, the *SOs* are the individual "augmented things" that together combine as a system to make *IoT* scenarios possible. Although the concept comes from the first decade of the century and the basis for the Smart Object-based *IoT* (*SO*b-*IoT*)

might be from the work by Kortuem, Kawsar et al.[69] back in 2010, nowadays the *SO*b-*IoT* and the *SO* as a concept are still under construction. There are various definitions and characteristics of the *SO* that vary slightly. There are also other terms that seem to refer to the same concept.

An *intelligent product* was defined, from a manufacturing perspective, as a commercial product with five characteristics: unique identity, communication ability, storage of self-data, a deployed language and decision-making capabilities [116]. Similarly, *smart devices* —including PDAs and mobile phones— were defined as physical objects with computing resources that are able to communicate with each other and with users [20]. Later on, *U-things* are described as physical things with digital abilities, as opposed to merely virtual *e-things*. The U-thing concept is a generalisation of *u-objects* —i.e. everyday objects —, u-spaces and u-systems[76].

Others coined terms including *Smart-Its*, *Spime* and *Blogject*. *Smart-Its* is an abbreviation of Smart Artefacts, i.e. objects that maintain their appearance and functionality, but are able to compute in the background [49]. *Spimes* refer to a more abstract entity, namely space-and-time-tracked objects that are dynamic and palpable although part of an intangible system [103]. Emphasising the role of objects as part of a social web, Bleecker [16] distances from “sci-fi" Sterling’s view and proposes *Blogjects* as objects that converse and exhibit location tracking, experience- holding, and the ability to foment action.

Yet, additional interpretations have brought wider definitions. In [13] an augmented entity is the composition of virtual and physical entities. In [84], the concept of *smart product* includes not only physical objects but also software and services. They are placed in a smart environment and gifted with self-organisation and interaction abilities that provide simplicity and openness. Later on, Gutierrez et al.[44] attempted to unify this denomination. They proposed a meta model with the *smart thing* as an abstraction of both smart product and intelligent product. However, they did not identify differences between these terms neither relations with other variants such as *SO*.

Smart object is probably the most common term to refer to the “augmented things”. It is defined from a technical perspective, as “an item equipped with a sensor or actuator, a tiny microprocessor, communication device and power source" [111]. Similar definitions can be found in [78] and [15], among other works. It is worth mentioning

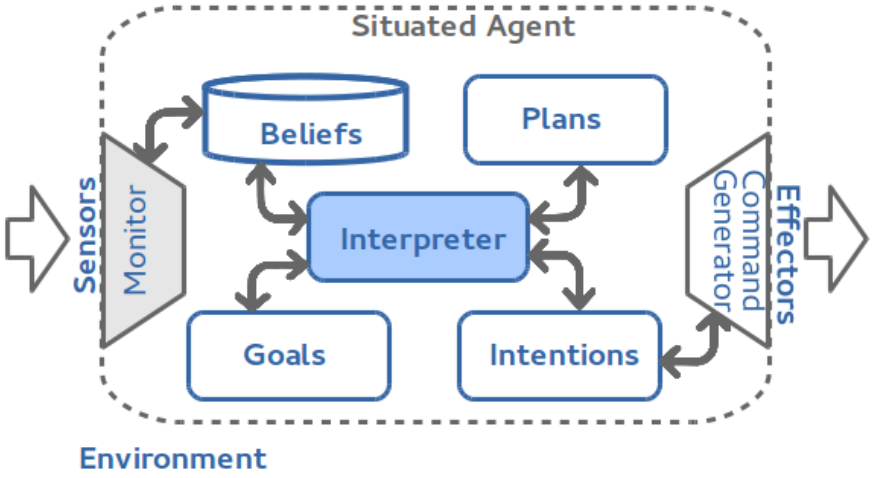


Figure 2.3: Procedural Reasoning System by Georgeff and Ingrand. Adapted from [40]

the work by Kawsar, who defines an *SO* as an object that "...augments human perception, is aware of its operational situations and capable of providing supplementary services..." [62]. This author also identifies key properties of the *SO*: Unique ID, Self-awareness, Sociality, Autonomy and State-fulness. In the following sections we present the fundamental concepts that we use to build the *SO*b-*IoT*.

2.3.1 Using agents for *SO*b-*IoT*

The agent paradigm provides powerful abstractions with direct mapping to the *SO* characteristics. An agent is defined by Wooldrige and Jennings as "a computer system that is situated in some environment, and that is capable of autonomous action in order to meet its delegated objectives" [117]. It is clear from this definition that the agent involves a software system that works autonomously towards some objectives. In order to meet these objectives, agents sense the environment, react to it, take the initiative to carry out actions, interact and work together with others.

The agent has an internal architecture that allows it to behave as described. Architectures are specific for every particular agent solution, but there are reference models that work as blueprint for ensuring the key processes and relations are considered. Although agent architectures are usually layered including elements of different models, we present one of the most popular and well-known models in order to show these processes and relations. The Belief-Desire-Intention (BDI) is proposed and applied, by

Georgeff and Ingrand, in the Procedural Reasoning System (PRS) [40] as presented in figure 2.3. PRS was originally designed for the operation of a robot but the concepts are of capital importance within the agent computing field.

According to PRS, the agent is not only able to sense the environment and react according to it, but also to reason based on its beliefs and commit (have intention) to generate and execute commands through the available effectors. Instead of having a complete fixed pre-programmed behaviour, the agent makes its own decisions based on the beliefs and the existing plans. Plans provide the set of steps required to achieve a goal. The interpreter is the central component that enables the decision-making based on the aggregated agent's knowledge. Details of each component are left open to be realised for each particular problem in multiple ways.

This internal view does not show agents are part of Multi-agent systems (*MAS*). In fact, some of the agent's plans might involve collaboration with other agents. For enabling this collaboration, the agents require a common language and mechanisms to communicate and coordinate with others in order to achieve their goals.

FIPA (Foundation for Intelligent Physical Agents)[3] is an organisation that produces standards for agent-based systems. The cornerstone of their specifications is the FIPA Abstract Architecture (FIPA-AA) [30], which gathers multiple contributions in the field as abstractions for building inter-operable MAS. Rather than a specific solution, FIPA-AA provides a conceptual basis for meaningful communication between agents that can be realised with different semantic and communication technologies. It defines an Agent Communication Language (ACL) that is used by agents in order to interchange messages. In addition, it defines the existence of a set of services that support agent's communication. The Message Transport Services (MTS) enables the exchange of messages between agents. These message's structure and transport are also defined throughout the specification. The other defined services are the Agent Directory Service (ADS) and the Service Directory Service (SDS). These are registries of agents and services respectively where every agent must register itself or its services and query for others or others' services. These registries are not part of an agent in particular, but are provided externally to every agent. Every agent depends on these registries to communicate with each others. It is assumed then that these services are provided by the environment where the agent are situated.

[3] http://www.fipa.org/

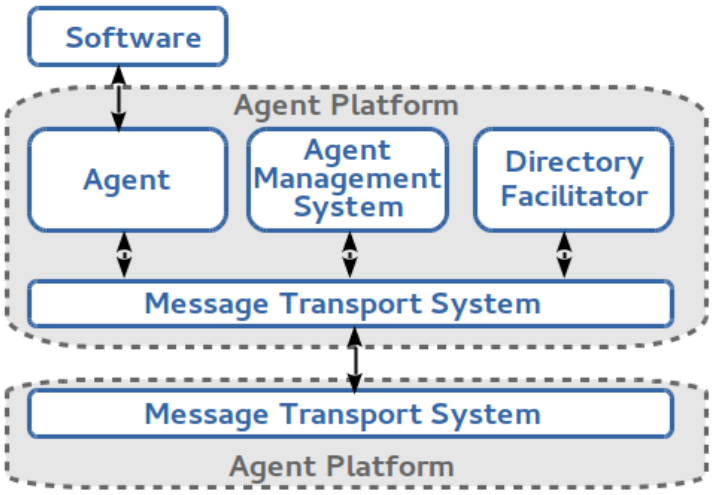


Figure 2.4: FIPA's Agent Management Reference Model from [31]

One example of environment for a software agent are the agent platforms. These provide the space where agents habit, communicate and cooperate. The most used agents platforms are based on the FIPA Agent Management Specification [31]. This specification is derived from FIPA-AA and defines a reference model (see figure 2.4) and the guidelines for managing and controlling agents within agent platforms. According to this specification, the agent platform is the physical infrastructure where agents are located, which might be distributed across several machines. The Agent is the actor with one or multiple service capabilities, one identity and one owner. The Agent Management System is the way that ADS is included in the platform, it is unique per platform and provides control over access and use of the it. If the ADS provides the white pages directory service, the Directory Facilitator provides the yellow pages one, enabling queries based on services offered by agents. The Message Transport Service is the communication method between agents. Finally, software is every non-agent routines that might cover other services, communication, security protocols and tools, among others.

From reviewing the FIPA specifications, particularly this reference model and the abstract architecture, important aspects can be noted:

- Although registries can be materialised in multiple ways, even as distributed or federated repositories that are replicated at different levels, conceptually, FIPA

conceives the location of agents and their services is based on the existence of centralised registries where the agents register to and query for others.

- Agents only exist within an agent platform, they are scoped and constrained by it where they are centrally controlled and supervised [31].

- Everything that is not part of the platform, nor the agents is treated as software, external to the platform and aimed to be accessed through the agents.

- Specifications have not been updated since 2004, leaving out any possible, support details, adjustment or guidelines for usage in new use case scenarios such as those in the *IoT* field.

2.3.2 Agent's goals

Goals have been widely analysed and described in the agent literature. We adopt the notion of goal defined by Sterling [104] that indicates that a goal is a "situation description that refers to an intended state of the environment" i.e. agents seek to provoke a target state in the environment. From this perspective, goals are expressed in terms of states that are measurable and can be discrete e.g. on or off, open or close; or continuous e.g. 20 Kg. These represent a high level way of defining the motivation of an agent. *Goal-driven behaviour* refers to the ability to guide agent operation by defining high level goals, instead of indicating detailed instructions like in an imperative programming style.

There are different classifications for type of goals the agents can have. One example refers to four big groups: achievement, cease, maintaining, avoidance and optimisation goals [104]. The first two refer to reaching or leaving (respectively) a target state. The next two types look at keeping agent's operation while maintaining/avoiding a defined state. And finally, the optimisation goals aim to minimise or maximise a defined function.

As an agent can have defined multiple goals simultaneously, there is a need to specify priorities among them. *Hierarchies of goals* enable the arrangement of goals according to its importance. These hierarchies are usually modelled as trees that are well known structures in computer sciences.

2.3.3 Agent autonomy

Autonomy is one of the core characteristics of software agents that makes them suitable to engineer software for *SO*b-*IoT* solutions. It is usually taken for granted that by using agents as building blocks for software the result is an autonomous system. However architectural decisions in agent platforms —the default frameworks for engineering agent-based software— and specifically in the solutions, constraint the autonomy. In this section, we examine the notions of autonomy in the agent literature that are the basis for the foundations of *SO*'s autonomy. Most of the foundations here described come from the works by Castelfranchi et al. [21, 22], Hexmoor et al. [47, 48] and Luck et al. [75]

There is not a unique understanding of autonomy. Different notions have been in the literature well before the beginning of the millennium (See figure 2.5). A common ground is that autonomy of an agent is a relational concept. It can be related to the agent's functions, goals and actions; to another subject or entity (something or somebody) [22] or to agent's own capabilities [47]. For example, an agent can be autonomous in triggering a process (action) in regards the human user (somebody), or it can be able to minimise the execution time for a process (goal) autonomously from the provider of the input of that process (something). If only the relation with itself is considered —the agent's internal view— the *absolute* autonomy is the ability to manipulate its own capabilities and determine what to experience from the outside [47].

In autonomy as a concept *relative* to other subjects or entities, there is a *social* and a *non-social* autonomy [22]. *Non-social* autonomy comes from the agent's condition of being situated in an environment. An agent is autonomous if it has its own goals and not only reacts to every environment stimuli, but it is able to perceive actively, select among these and decide whether to respond or not and how to do it, based on its own internal states. In other words, the environment (at least not exclusively) does not drive agent behaviour [22]. This definition helps to clarify why autonomous agents are different from objects[4] and service providers, the latter ones do not have control of the decision to carry out the methods or services, according to the case, and just react to calls [117, 104].

Social autonomy comes from the relationship of the agent with other software agents or

[4] the notion of object from the object-oriented paradigm

the human user. It is conceptualised from the views of *dependency* and *collaboration*. The most common and simple conception is that of autonomy as the ***independence*** —i.e. the agent does not need intervention— from the human user for a particular goal, function or action. Under this view, it is worthwhile to differentiate autonomous from *automatic* agents, where the latter ones have an *unreasoned* —they are not able to make decision and reason— behaviour for a particular task, exhibiting only some degree of autonomy, in contrast to the former agents that do not hold these constraints. The collaboration view is more complex, it is defined in the context of some goals, functions or actions that are delegated to the agent. We simplify Castelfranchi [21] defining autonomy in collaboration as the ***multilevel characteristic*** of an agent within a context, determined by the existence of its ***own goals***, the ***discretion*** to make decisions —including following rules or not—, the ***initiative*** for engaging in tasks, the effort required for ***understanding*** these, and the ***control*** it might have in producing the results, monitoring the progress and executing the necessary steps for achieving these.

Considering how the goals, functions and actions determine autonomy, it is defined at three levels. At *executive* level, the agent only decides on the execution of a plan; at *planning* level, the agents is allowed to plan; and at *goal* level, the agent has its own goals [21]. In contrast to having multiple levels of autonomy and looking at concrete functions, Luck et al. [75] propose another view of *absolute* autonomy as a property with a binary discrete domain: the agent has the ability to ***generate its own goals***. Other authors [82] adopt the same absolute position, but see ***decision-making*** as the key function that determines autonomy.

Finally, considering the variability of the autonomy, it could be fixed or adjustable. From a multi-level autonomy, the concrete level of autonomy for an agent can be *fixed* or *adjustable* by the subject who delegates the task. The adjustment of autonomy could be done before runtime, during runtime and after execution (affecting only future behaviour) [22].

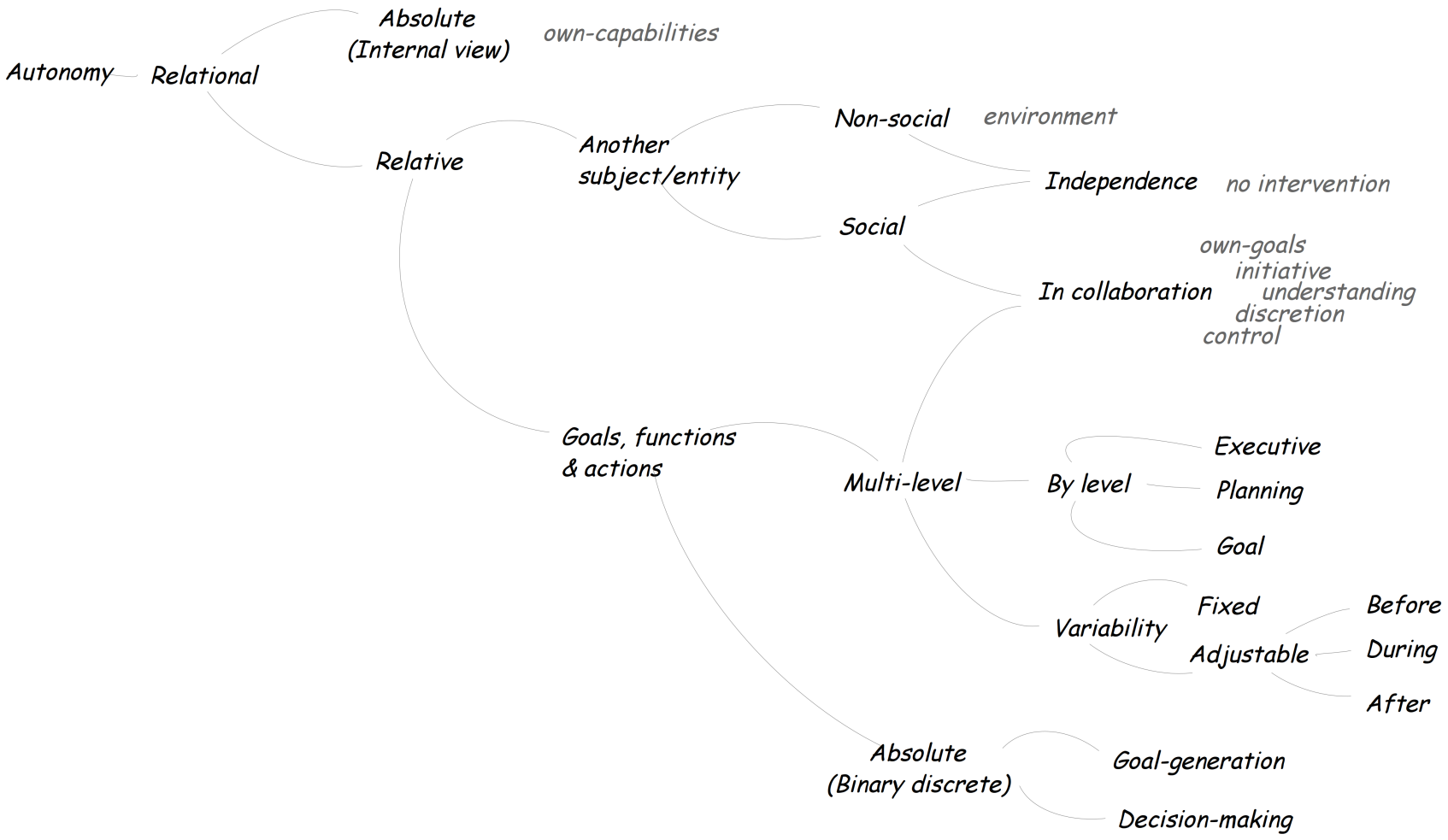


Figure 2.5: Notions of autonomy from the literature

Autonomy and Architecture

The autonomy is derived from the agent's architecture as the latter provides the resources and capabilities the agent uses to accomplish its goals [21]. The architecture is an enabler of internal power such as knowledge, skills and capabilities. If the agent does not have the required power, it has to depend on external power such as material and social resources. Hence, the architecture might generate by default dependencies on another subject/entity if it is not designed to provide what is required to achieve the agent's goals.

These ideas are still valid with little fundamental changes to the core concept as shown in a recent survey focused in adjustable autonomy [82]. We have seen the concept of autonomy is abstract and with multiple views. In order to ensure that requirements of autonomy are considered in the software for *SOs* and that the *SO* is designed without hampering it, the concept needs to be more concrete and applied to the context of interest. Little attention has been given so far in *SO*b-*IoT* literature to analyse autonomy in *SOs* and differentiate it from the more abstract agent autonomy. We provide a more concrete view in section 3.7 adopting independence from another subject/entity as the basis of our definition.

2.3.4 Joint Use of Agents and Web Services

Service and Agent computing paradigms have proven to be powerful and useful software building blocks in multiple contexts. For *IoT*, this is not an exception. We have said that web services are particularly suitable for web application development as they provide inherent interoperability and reusability (Section 2.2). On the other side, agents and *MAS* are instrumental in development of autonomous and cooperating systems that are able to reason and proactively take actions. Inter-operability, autonomy and cooperation are fundamental for achieving the *SO*b-*IoT* vision. However, it is a challenge to ensure these paradigms are used together effectively, enhancing each other and taking advantage the existing common grounds.

Agents and web services have traditionally been treated as separated worlds requiring some integration efforts. This has resulted in the perception, from a technology point of view, that these worlds can not interact with each other directly [39]. Looking beyond what current solutions offer, we identify three conceptual approaches for work-

ing with agent and web service environments.

- *Web-integrated agent services*
  When agent platforms are not wide open to internet these require integration and therefore an interface to it —See figure 2.6(a)—. The interface enables on one side that agent services within a platform consume web services and on the other side, that agents services can be exposed as web services, so non-agent-based applications can use them. The interface is in charge of translating messages from one world to the other. Several integration efforts have been reviewed in [42]. Some are either ad hoc solutions as the ones proposed for different domains in [127, 14]; or platform offered as the gateway and dynamic client add-ons provided by the JADE Platform[5] [52, 51].

- *Agent-backed web services*
  Whereas the previous approach targets individual agent services, in this case an entire MAS is wrapped or used for generating web services. This could be the case of a MAS mediating for service composition as in [96, 45]; or that separated, unconnected MAS inter-operate through web services [46, 59]—figure 2.6(b)—.

- *Web Agents*
  In this approach the agent environment is the internet —See figure 2.6(c)—. Services define the behaviour of the agent, they allow to benefit from the resources and infrastructure already existing on the Internet. Agents do not require gateways, intermediate agents or services to communicate with others via web services. This approach is less exploited, one example of an agent platform available is [56] and a conceptual view describing a similar approach, as described, is found in [54].

We think Web Agents is the best approach for realising the *SOb-IoT*. Not only because it provides direct access to web resources, infrastructure and communication through standard protocols enabling the inter-operation of heterogeneous platforms as *SOs* are, but also because it does not enclose the agents in a controlled environment. Hence, this approach has the potential to enable software engineering based on agents that are not limited by the platform therefore more autonomous than traditional FIPA agents.

[5] http://jade.tilab.com/

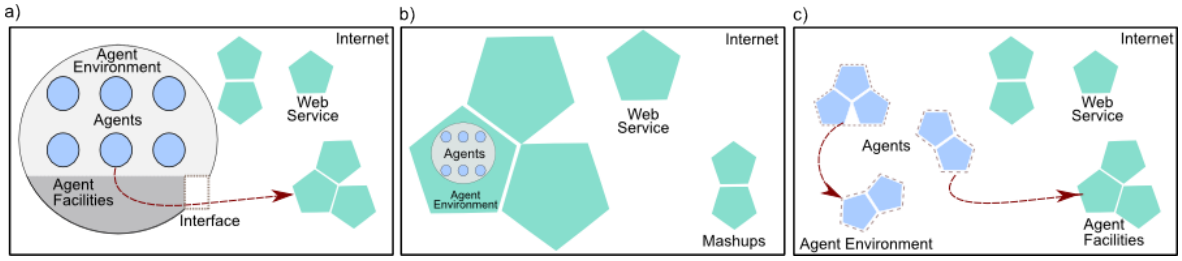


Figure 2.6: Joint use of Agent & Web Services: a)Web-integrated agent services b) Agent-backed web services c) Web Agents

2.3.5 Autonomic Systems

Since *SOs* must exhibit self management characteristics in a highly heterogeneous and dynamic environment, they must behave as autonomic systems. These are systems able to adapt to frequent changes and new conditions at runtime, minimising human intervention [71]. In *IoT*, the focus of this work stream is to diminish the effort required when managing *IoT* applications by automating all or part of the tasks related to hardware, software, network, sensors configuration in order to connect to, read, publish or process data sensed from the several dissimilar devices.

The fundamental features of autonomic systems were initially identified by IBM and then summarised in different publications [65], these characteristics are:

- *Self-configuration*: The system is able to establish and modify configuration parameters according to high-level policies.
- *Self-healing*: The system is able to recovery from failures occurring at runtime.
- *Self-optimisation*: The system is able to pro actively change parameters in order to improve performance or efficiency.
- *Self-protection*: The system is able to detect and anticipate to threads taking proper actions to ensure security and integrity.

One of the most popular architectures for autonomic systems is the MAPE-K Model also defined by IBM [64]. This defines autonomic managers that are goal and knowledge oriented, able to perform the processes of monitoring, analysis, planning and execution over a set of managed artefacts. Within these processes there are two particular key functions that are relevant for this work: the adaptation and decision function.

The adaptation function covers how the system modifies its structure and behaviour

in response to changes in the environment in order to keep working towards the defined goals [71]. It comes from modifying code, data or resources and can be done at operating system level, at program or component level [71]. The first case includes the services provided by the *SO* to deal with configuration of new hardware resources and common libraries, dynamically on-demand. The second case, is the adaptation achieved by programming languages within an application, E.g. using interception or dynamic linking when the language supports it. The last case, gives support to changes at high-level based on programmed building blocks from which the application is built. It includes mainly components but also services or the hybrid service-oriented components [23].

On the other side, the decision function is about how autonomic systems represent and use the available knowledge to make decisions. There are several approaches mainly coming from intelligent systems literature [71]. Techniques include rule, goals and utility-based systems, among others.

The different functions of the autonomic systems can be carried out centralised or decentralised. Decentralised mechanisms are particularly challenging because decisions are made with partial information and coordination mechanisms are required between the different nodes that are part of the system. There are scarce practical approaches that offer decentralised solutions and the drivers for deciding when to use decentralised control loops in self-adaptive systems are not clear yet [27]. The *IoT* environments with their heterogeneity, volatility and resource-constrained nodes, offer a rich set of concrete situations where decentralised self-adaption solutions come to their play.

2.3.6 Role-based architectures

The role concept is embedded in the view of the systems as organisations. These abstractions enable the modelling of highly dynamic and adaptable systems where the individual parts —role players— and the whole —the organisation— can vary quickly and easy. Roles have been used extending both object-oriented paradigm as in [70] and agent paradigm, as in the works surveyed in [19] and the others presented in [121, 104, 126]; or as an independent paradigm as in [25]. Main advantages of these abstractions include:

- Facilitate the definition of the system' behaviour as the processes, functions, components of one organisation that can be reused and adapted.

- Enable the definition of individual functions of components of the systems or subsystems uncoupled from the concrete component.

- Provide a model for coordination between components or subsystems of a system.

Given the powerful advantages this abstraction provides, it is surprising that it has been rarely used in the development of *SO*b-*IoT* systems.

The definition of a role varies according to the properties and scope given to it. For example, Colman identifies a *Player-centric* viewpoint and an *Organisational-centric* viewpoint [25]. In the former one, players are stable and roles are attached/removed from them. Roles do not have identity beyond the players but are only an appearance of them. In the latter one, the identity and existence of the role is derived from the organisation [25]. There, the roles also define the processes carried out within the organisation and the players are executors of it. It is clear, that latter view includes a broader more complex notion of role.

We think neither of the mentioned views suits the context of *SO*b-*IoT* applications well. In the first case, because roles might exist and have identity even if there are no players available. Players —SOs— are highly dynamic in *IoT* scenarios coming in and out very often, in addition the characteristics that cause the role-assignment, might also change, so in one moment the *SO* suits the role and the next not, regardless of the originally attached roles. On the other side, having a complex definition of role makes it a more sensitive entity. In particular, a process definition is, itself, complex and it may change independently of other role properties. By adding the process to the role, this complexity is shared as well adding multiple sources of change to the role. As *SOs* are resource-constrained, the idea is to take advantage of the role abstraction but minimising the management required, for that it is required that role definitions be as stable as possible.

For the mentioned reasons, we adopt a simpler role definition given by Sterling. He describes a *role* as the capacities required to achieve goals [104]. On top of that definition, we pick from the literature [70, 102, 25] the role characteristics and adapt them to our role definition:

- *Visibility*: Object (player) is visible and accessible through the properties of the role.

- *Multiplicity*: A role can be played by multiple players.
- *Dynamics*: Roles can be added or removed during player's lifetime and roles can change their definition.
- *Normative*: The assignment of a role to a player indicates its responsibility to carry out the actions described on it.

Management of the roles is a derived management task of working with them. Colman defines a functional role as that related to a functional process of the organisation and an organisational role as that related to the organisation management including roles [25]. The organisational role players are seen as meta-players that create and destroy roles among other tasks.

2.4 *SO*b-*IoT* Middleware: State of the Art

The main related work of this thesis is that of middleware solutions that intend to provide facilities for both *SO*b-*IoT* software development and *SO* operation.

2.4.1 *UbiWare* Project: Middleware for Industrial Systems

UbiWare [60] was a project for building a platform for development of industrial complex systems. It was built from the SmartResource Platform [61], their previous effort in a development framework for MAS. In *UbiWare*, they followed an approach based on the view of a Global Enterprise Resource Integration (GERI) system, that enables not only inter-connectivity but also inter-operability of different resources that are linked to it. Resources may include physical devices, web services, software applications and even humans. They proposed various industrial use cases, for example: *UbiWare* for connecting multiple systems (E.g. intelligent, experts) for detecting failures in power networks or UbiWare for automation of Service Desk operative processes in a Telecom Operator.

Through the definition of a software agent, representing each resource within the platform and a semantic adaptor, these resources become smart. These are proactive and self-managed resources able to control their own state, communicate and coordinate with other resources. They give special emphasis to the role of semantic technologies as enablers of the coordination, discovery and common understanding between each resource.

The architecture of SmartResource and *UbiWare* is built on top of JADE framework and is composed of three main layers: Reusable Atomic Behaviours (RAB), Behaviour Models and a Behaviour Engine. A RAB is a "piece of Java code" [61] that implements an atomic function. The behaviour is defined by representations of organisational roles which are linked to documents defined using a Semantic Agent Programming Language (S-APL) which is based on RDF[6]. The behaviour engine which is part of every agent and provides their default behaviour, consisting on parsing the RDF-based document, registering the role with the Directory Facilitator and the agent life cycle. Additional behaviours are obtained from a external repository, that is supposed to be managed by the organisation.

Although *UbiWare* is one of the first attempts to introduce agents to endow human-related autonomy to the *IoT* devices it does not incorporate the concept of *SO*. In addition, it relies heavily in the GERI platform for the different interactions between resources. The existence of a central behaviour repository gives control over the definitions however it is error-prone and impose exogenous dependencies in the agents managing resources.

2.4.2 *FedNet*

FedNet is a document-centric framework for building *SO* systems [63, 62]. This is one of the first platforms to be designed under the notion of Smart Object. Authors of Fednet considered that *SOs* had multiple roles, however the proposed software programming model was not based in this structure but in service profiles. SOs expose their features as documents as the applications expose functional tasks that need to be carried out by *SOs*. Application developers use these documents regardless of the actual *SOs* that carry out the task. There is an intermediate infrastructure that connect applications and *SOs*. The infrastructure carries out bootstrapping, management and enable utilisation of *SOs* independently of the applications.

There is an *Artefact Framework* component which is the digital identity and encapsulates an *SO*. This component can be deployed "at-edge" on the artefact, or "at-infrastructure" in a proxy, which is powerful device (e.g laptop). Its logical architecture is composed by a core and a "cloud" of optional smart features. The core is common to

[6]Resource Description Framework (RDF) is a data model

every *SO* and contains modules for communication, notification, local memory, a client handler and a plug-in repository that manages the smart features. This design allows to have objects that although physically identical can differ in the smart features offered. These features are called *profiles* represent generic services that do not depend of the artefacts.

FedNet works as a Gateway connecting applications and services using RESTful service calls. It is composed of: an *Application Repository*, *Artefact Repository*, the *FedNetCore* that generates templates of federation of artefacts that are attached to *Access points*. *Access points* represent the physical environment needed by the application and are delegated by these to forward requests and receive responses from artefacts. Components of FedNet can reside in different nodes along the network.

2.4.3 *ACOSO*

The Agent-based COoperating Smart Objects middleware [32, 33, 35] is perhaps one of the solutions most aligned to the *SO*b-*IoT* vision. Their approach is based on the view of *IoT* as a system of decentralised and cooperating *SOs* and ACOSO provides a programming model for it. *SO*'s hardware structure include wireless sensors/actuators and either a PC, notebook, tablet, smartphone or embedded computing device [33], whose power is pre-defined according to the functions it will accomplish. The case study proposed is based in a smart office scenario, where the *SOs* recognise activities and location of the user to active some behaviours.

ACOSO is agent-oriented, event-based and its architecture is based on a general master/slave model. In this general architecture the computing device is the *SO*'s master and the sensors/actuators are slaves [32]. Besides, it comprises functional blocks for managing: input/output, communication, discovery, knowledge base, context, sensors and actuators. It relies in the notion of a Directory Facilitator which enables the discovery of *SOs* using different criteria E.g. services, location and different *SO* properties.

Instanced from the general architecture, *ACOSO* architecture is organised in three subsystems and two components. Subsystems are for communication, knowledge base and device management. The Behaviour component is a set of event-driven system and user tasks that intend to reach specific goals. The Event Dispatcher is a central component that manages a queue of events related to these tasks.

From this architecture, they offer a middleware implementation and application addressing their key case study. Their implementation is based mainly on JADE but also in Jadex[7] and MAPS —a previous effort of the same group [3] —. Tasks are defined as Jade behaviours or Jadex plans, the execution is provided by each framework.

2.4.4 *ASAWoO*

ASAWoO: Adaptive Supervision of Avatar/Object Links for the Web of Objects [54, 83, 107, 66, 106, 79] is a big project that aims to build an architecture that provides functionalities for enabling the creation of WoT applications, in which, from basic sensors to complex robots, are able to collaborate with each other. They envisioned a Cyber-physical object made from a physical object and an *avatar* —"virtual extension of physical of physical objects" [83]— i.e. an agent able to communicate and cooperate. Their platform is intended to be installed in powerful devices (PCs, laptops) or in the cloud to support constrained ones. The platform is required to be inter-operable, adaptable to environment, able to delegate and determine objects that can perform actions based on estimations of usage, computation and networking; and tolerant to disconnection. Their approach is mainly based in RESTful web services, web semantics, the OSGi architecture[8], disruption-tolerant protocols and using agents for task allocation.

Capabilities are described semantically enabling the platform to infer which are required for a given high-level functionality –E.g. –. A multilevel context model and processing engine that determines where to deploy code, which protocols to use, which functionality to perform and whether to collaborate with others or not. For disruption tolerance they work with routing protocols based on both centralised and distributed services discovery. The autonomous behaviour is based on interaction situation by the avatars that determine to be indifferent, cooperative or antagonist to others. The functionalities every device offer are exposed as RESTful resources and stored in a *Functionality Directory* in order to determine objects to execution application.

The architecture is grouped in the following functional modules:

- *Core* includes a reasoner and functionality, deployment and context managers.

[7] https://www.activecomponents.org
[8] Open Service Gateway initiative: https://www.osgi.org/developer/architecture/

- *Web service* exposes single-object or multiple-object functionalities as applications. It enables exposition of object's services directly as RESTful services that can be called by other avatars.
- *WoT application* provides description of the object behaviour to end-user
- *Local functionality* carries out introspection of capabilities and deduction of functionality.
- *Collaboration* Looks for other object's functionality and negotiate with them to expose it in the WoT application server.
- *Communication* selects the right communication protocols.
- *Filtering* makes context-based decisions in regards which functions to deploy where and protocol to choose.
- *Object Inter-operability* connects to repositories of drivers and files to be able to configure the object.

The existence of a Gateway or a cloud platform is assumed as it enables constrained devices to work. This is required to host an avatar builder that detects incoming objects and creates new avatars. Applications are deployed in the platform and a semantic process decides which objects to use and when to use the cloud. When writing this section, the project was in its final stage with intensive and important contributions reported separately, although the operation as whole is not clear, E.g. how the agent-based task allocation mechanism works over the disruption tolerant protocol.

2.4.5 Leppänen

A lightweight approach is presented in [73, 74]. They aim to distribute the processing of data among the *IoT* devices. Their approach is based in the composition of *Mobile Agents* that run atop *SO* platforms.

Although they call them mobile agents, it is not clear what is the difference of these agents to the standard web services. As these are accessed via REST interfaces, it is not mentioned if the agents have some autonomy to process or not a given request. A resource directory is the basis for localisation of services and communication relies on HTTP and COAP. The composition of mobile agents is made from key segments that include information about the code of the task to be done by an agent, the resources needed for the task and state of the agent.

Every *SO* has an architecture composed by the physical components, an execution envi-

ronment, an agent and object interface and a repository. The execution environment is the central component in charge of running the agent's tasks based on resources stored in the repository. It also carries out configuration of *SO*'s physical components and controls the lifecycle of the agents. A series of agent interfaces including marshal/unmarshalling, execution and stopping, among others enable to control the agents. They also provide REST interfaces for communication between objects and also relies in the existence of a gateway to interact with "external" services.

2.4.6 Other platforms

It is worth mentioning a few additional middleware solutions. Jung et al. [57] present an Agent Service Platform, mainly addressing the management of heterogeneity in devices. Resource-constrained devices based on Arduino, delegate control to agents placed outside them, but able to communicate with other agents and devices through a message bus. Runtime adaptation is possible using Portable-Service abstractions and dependency-injection patterns. Agents in this proposal are located away from the object they represent and depend on an external agent manager.

UbiComp [41] offers a programming model and middleware for development of ubiquitous applications based on the composition of artefacts (*SOs*). The focus in the programming of such applications using visual editing tools. They enable that *SOs* locate each other based on the service descriptions. A visual platform enable to link different artefacts trough "synapses", carrying out for example measurement, reasoning and actuating functions. They do not address the problem of having multiple *SOs* offering the same service nor a goal-directed behaviour to the *SO* which is required to endow autonomy. *UbiComp* and other solutions have been surveyed and compared in [34].

Most of the works we have reviewed deal with autonomy as an abstract concept, considering only the human user perspective and under the assumption that using agents as part of the development implies an autonomous behaviour. In most of the cases, web services and agents are considered two separated worlds that require additional integration efforts. Likewise, social characteristics of the *SO* are barely considered along *SO*b-*IoT* middleware solutions. In our evaluation (Section 7.7), we provide a further comparison of the main solutions from the autonomy point of view.

2.5 *IoT* Service Selection

Since service selection is central part of our adaptation approach, we turn now the attention to *IoT* service selection solutions. We note that service selection has been a very active topic in the research community. Attention is centred in distinguishing and comparing services based on other criteria than functionality. Several studies address the key questions of how to describe the non-functional properties and the methods for choosing the most suitable given some requirements [100]. A thorough survey of service selection based on non-functional properties is presented in [122], presenting a classification of the different approaches mainly based on the identification of *QoS* attributes.

Common strategies for selection include taking advantage of service semantics and metrics to enable objective comparison given a defined criteria [95]. Recently service selection has been approached from the *IoT* field, the main interest has been around incorporating physical properties to the selection criteria and identify methods to assess these properties. [67] presents an energy-centred approach that intends to maximise the availability of the *IoT* devices of an application. This approach uses pre-selection and a method based on lexicographic optimisation of individual *QoS* attributes. Energy *QoS* are used to establish a relative dominance criteria that enable to establish a total order relationship between the pre-selected candidates [67]. In [55] a model of physical service properties is presented that enable the definition of *QoS* attributes. Their selection is split along design and run time and is based on the individual *QoS* attributes, a ranking generated from them and a absolute dominance relationship defined for the importance of some attributes to the user. Since these approaches do not contextualise the service selection methods within a complete end-to-end *IoT* scenario, they do not consider practical aspects —e.g. a big quantity of *QoS* attributes — that can complicate the preference definition.

[7] proposes a dynamic service-arbitration scheme for *IoT* systems. This scheme enables the selection of a reduced set of devices to be active when there are multiple providing equivalent services. A multi attribute decision making problem is formulated with energy used, sensing frequency, number of neighbours and memory resources are the representative attributes. They use the Technique for Order of Preferences by Similarity to Ideal Solution (TOPSIS).

There are solutions identifying the need of selecting *IoT* devices to perform specific tasks within a system. [107, 83] introduce an approach for multi-purpose adap-

tation addressing the selection of appliance's local capabilities needed for a high-level functionality. The *IoT* device's capabilities are described semantically enabling the platform to infer which are required for a given high-level functionality. They present a multilevel context model and processing engine that determines where to deploy code, which protocols to use, which functionality to perform and whether to collaborate with others or not.

Another solution that offers a sensor-centred approach is CASSARAM [89]. Authors propose a model for search, selection and ranking sensors based on user priorities. Their idea is that users enter manually their preferences on a broad set of sensor properties including reliability, battery, precision, among others. This platform works effectively in ranking and selecting from a large numbers of sensors with overlapping and redundant functionality.

A common characteristic of the solutions reviewed in this group is that these assume complete knowledge of the connected devices, services and their attributes or *QoS* properties, so these are conceptually based on a central repository gathering all network information, which is unpractical for autonomous *IoT* solutions.

2.6 Summary

The IoT is a novel paradigm that brings several disciplines together in order to enable enhanced physical objects to be connected and interact with each other for the benefit of the human user. The foundations of this paradigm are still under construction. There are different realisation models identified so far, with the "data-feeder" approach as the popular choice for IoT application development. This approach is based in the existence of a platform that centralised the application logic and controls the different connected devices that behave as mere data feeders.

The *SO*b-*IoT* approach envisions the IoT as a collection of smart objects that as a whole make IoT applications possible. Although there is not agreement nor on the characteristics or the concept definition these objects, they work as a building block for IoT applications. Software agents and web services are two popular paradigms used for building *SO*b-*IoT* software systems. However, there are different challenges in regards how these are applied.

Autonomy is a key *SO* characteristic although contradictory, little attention has been given to convert abstract notions into concrete ones, for the field. There is a mis-

conception that simply because *SO* software is based on software agents, it becomes autonomous. Autonomy is a relative concept that can be analysed in relation to other subjects or the goals, functions and actions. It is also challenging the way that agents and services are used, traditionally they are seen as two separated worlds although *SO*b-*IoT* requires a more effective and blended approach. Part of the autonomy involves the capacity to self manage, i.e. behave as an autonomic system. Roles is a powerful abstraction, still rarely used in the development of *SO*b-*IoT* systems.

We have reviewed some of the most important *SO*b-*IoT* middleware and *IoT* service selection solutions. The most of middlewares surveyed work with autonomy as an abstract concept and do not consider the implications of the design decisions on it. In fact, most solutions rely in a platform that provides services to distributed *SOs*. For a true *SO*b-*IoT*, an approach for building software and tools for enabling it based on a defined concept of *SO* autonomy are required. From the side of *IoT* service selection, in most of the cases there is a lack of decentralised solution that does not assume complete knowledge of connected nodes and services.

Chapter 3

Foundations of Smart Object's Autonomy

3.1 Introduction

Considering the variety of concepts and view points around Smart Objects (*SOs*) and autonomy, the aim of this chapter is to provide the conceptual foundation for Smart Objects, their autonomy and the role they play in the *IoT* software engineering. It seeks to enable scoping of the *SO* term and identification of the relevant elements for the analysis of the *SO* autonomy. In this chapter we built from existing literature (Chapter 2) and describe *SOs* and collectives of *SOs*. We propose an analysis tool for describing *SOs* in detail.

After identifying main challenges (Section 3.2) and contributions (Section 3.3) of this chapter, we present a summarised description of the *SOs* and collectives of *SOs* (Section 3.4). Next, we introduce the proposed analysis tool (Section 3.5) that brings together characteristics of an *SO*, envisioned as an autonomous system. The succeeding section 3.6, presents the examination of the *SO* characteristics according to the analysis tool. It is worthwhile mentioning that there is a wide spectrum of *SOs* that do not exhibit several of the characteristics identified, as these represent a vision of the *SO*.

Finally, from the examined characteristics we examine and discuss the concept of *SO* autonomy in section 3.7. We analyse how the lack of some of the *SO* characteristics have impact in its autonomy. In this regard, we identify the scope of the *SO* autonomy and describe some of the main dependencies that might hamper the *SO* autonomy.

3.2 Research challenges and requirements

The vision of the *IoT* considers the *SO* as autonomous systems. These are complex active entities that can be analysed from multiple perspectives, each one according to different interests. This chapter aims to cover the following challenges:

- Understanding of Smart Object's Autonomy
  Since *SOs* are embedded in a Cyber physical environment, they are not merely software agents. The physical dimension and the localisation of the software functions have impact in the autonomy and these are aspects to be considered in the *SOs* software engineering. It is necessary to define the concept of *SO* autonomy, what are its boundaries and what are the obstacles that might prevent an *SO* to fully develop it, so these can be avoiding when designing *SO* systems.

- Smart Object Analysis
  In order to analyse autonomy, it is necessary to understand the entity of interest, its features and scope. A thorough analysis of the *SO* is then required, starting from the concept of what is and what is not an *SO*. It should determine how autonomy can be analysed in such heterogeneous entities and what are the *SO*'s elements that have impact on it.

- Approach for an organised analysis of *SO* systems
  It is difficult to identify a proper and complete set of elements of analysis, specially when the subject under study is the construction of multiple disciplines. We observe the lack of normalised analytic tools that target the *SO*-based systems in order to enable the identification of their relevant features, their relationships with the environment and the distinction from other existing entities. This absence hampers the applicability of the advances made in one field to others. An analysis tool might help to drive the research efforts through equivalent areas and characteristics, reducing the overlapping efforts.

3.3 Contributions

We describe the main contributions of this chapter:

- Definition and analysis of the concept of *SO*'s autonomy, derived from the agent's autonomy and relative to *SO*'s key properties and its relationships with the environment and other relevant entities, rather than an absolute feature.

- A systematic analysis of the *SO*, its properties and behaviour as an autonomous entity from a system perspective and considering its individual nature and its role as part of a collective of *SOs*.
- As derived contribution, we propose a schema that organises the analysis of autonomous systems. It facilitates the analysis by providing a set of views covering key areas, fundamental processes and planes that lead the thinking towards the key aspects that characterise an autonomous system with a holistic approach. The schema provides an homogeneous reference of analysis for cited systems and related solutions. Therefore, regardless of the different characteristics of the subjects of analysis, these can be organised and compared following a common set of elements of analysis.

3.4 *IoT* Autonomous Systems: Individual and Collective

When developing *IoT* software systems there are two levels to be considered: individual and collective. At individual level, each Smart Object (*SO*) is an unique entity and a system that works with a defined purpose. It has responsibilities that lead to the achievement of the purpose by using its capabilities, being aware that is placed in a environment and surrounding by other systems. On the other side, in many *IoT* scenarios, one purpose is common to multiple *SOs*, so they can cooperate in order to extend each other's capabilities. In the case of cooperative *SOs*, when capabilities of individual *SOs* are not enough for achieving their individual purposes, a collective of *SOs* cooperates in order to achieve a common purpose. Individual *SOs* and collectives of *SOs* face different challenges that need to be considered when engineering *SO*'s software systems.

At the individual level, the *SO* has to deal with situations such as:

- Executing individual plans leading to own goals.
- Overcoming unexpected situations such as lack of data, failure of hardware components.
- Configure and upgrade its components.
- Switch on and off its components as per defined goals.

At the collective level, a set of *SOs* has to deal with situations such as:

- Executing cooperative plans leading to common goals achievement.

- Overcoming unexpected situations such as departure and arrival of *SOs*, changes in available *SOs*.
- Latency and noise in communication channels.
- Select best *SO* available to carry out a task.

Both a single *SO* and a collection of them might behave as autonomous systems. A summarised description of *SO* and collective of *SOs* gives an overview of the key characteristics of each one.

3.4.1 Smart Object

Definition 1 *A Smart Object (SO) is an autonomous cyber physical system (CPS) built from an ordinary physical object that is enhanced with digital and social capabilities. The* SO *maintains the object's original essence but is an active entity that exhibits autonomous behaviour.*

Key characteristics

- A unique identity
- Relevant knowledge to carry out its operation
- A motivated behaviour, *SOs* have a purpose and they are aware of it.
- A set of core capabilities that are subject to be updated and/or enhanced.
- Location in a cyber physical environment.

An *SO* is also a holon as it is composed by other sub systems and is part of a broader system. An *SO* is an active subject that interacts with other subjects including other systems, human individuals and collectives of them. There are many types of *SOs*, ones with a richer mix of capabilities than others. The smartness of the *SO* is associated to its autonomy from different perspectives. A Smart Object is different from Smart Thing in that the latter one is more abstract and includes a broader spectrum of devices that have some of the characteristic of *SOs* but not all of them. e.g. An object with cyber physical features such as sensors and actuators but that has no motivated behaviour; or a smart room with multiple actuators as well as data processing and sensing units. In this work, we concentrate in the *SO* as a whole entity which interacts with one or multiple human users.

3.4.2 Collective of Smart Objects

Definition 2 *A collective is a system of* SOs *that are connected and interact with each other. The collective is a society where* SOs *have responsibilities and a position.*

Key characteristics

- It is built around a defined criteria, E.g. proximity, common ownership, functional domain, supply chain stage, etc.
- It has knowledge, resources, relationships, structure and behaviour, that by default, is an aggregation of the individual *SOs* that are part of it.
- It is built from heterogeneous or homogeneous *SOs*. The heterogeneity comes from the particular mix of *SO*'s functionalities, their individual properties —E.g. location— and the properties of the functionalities offered by the *SOs*—E.g. quality, performance, etc.—.
- Within a collective, both cooperative and competing behaviours can emerge from the *SOs* that are part of a collective.

In the scope of this work we only deal with collectives where a cooperative willingness is assumed. This means that *SOs* are willing to cooperate with others as long as they have the resources and capabilities to do so. In this work we do not address scenarios with competitive behaviours by the *SO* collective members.

3.5 *E-Ma-Gen3* Framework: An analysis tool

In this section, we present the analysis tool we defined to examine the smart object's properties in detail. *E-Ma-Gen3* allows to analyse, conceptualise, describe and "imagine" *SOs* as autonomous systems from three perspectives as shown in figure 3.1. The analysis can be started from any of the three perspectives and the findings structured in a scheme, as the one presented in the mentioned figure, that can be read from each key area, process or plane. The three combined perspectives of analysis are:

- **Key Areas**: The columns aggregate elements of analysis of the subject in regards Knowledge, Capabilities, Relationships, Structure and Resources. The areas are static points of reference from which the analysis is approached.

- **Fundamental Processes**: The rows provide a dynamic view, identifying the set of processes the systems carry out as part of their operation. These are broad

processes that are carried out from elements within each area and include: Exploitation, Management and Generation. Processes are made concrete by the capabilities available in the system. Processes increase in complexity from bottom to top.

- **Planes**: These are the source of rules, restrictions and details that have impact in the system operation. Planes provide context and a reference for scoping of the subject of analysis. Although these are analysed under each key area and process, it is the behaviour area that mostly defines the scope of the system within the planes. Planes are particular to the field of study.

Every system has an initial set up and state for processes and elements under each area. The system evolves from the initial state by carrying out the fundamental process, ending up in a resulting state. That resulting state determines the complexity of the system.

E-Ma-Gen3 was conceived to guide and provide structure to the examination of Smart Object systems. The tool provides support to organise thoughts and ideas around the cited systems. When examining *SOs*, the tool is intended to help to differentiate between existing *SO* solutions and determine how far these are from the *SO* vision. It also works to scope the research efforts in the multidisciplinary field such as *IoT*. The processes, key areas and planes identified in *E-Ma-Gen3* are general enough to support analysis of other autonomous systems. In fact, the *SOs* are a type of autonomous system. Therefore, there is no evident restriction of *E-Ma-Gen3* to support the analysis of other type of autonomous systems, however this is no evaluated as this is not part of the scope of this work.

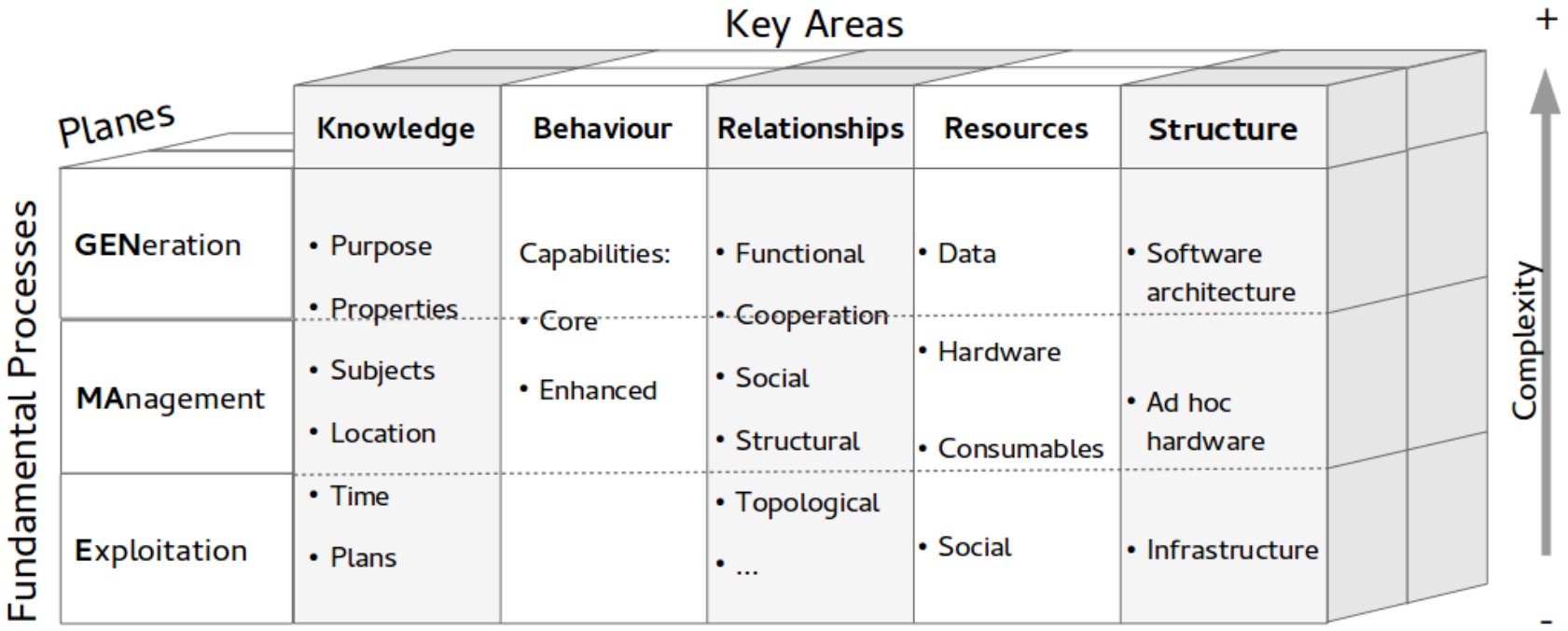
The definition of the tool emerged from the need to identify and analyse the relevant characteristics of the *SOs*. There were not available guides to carry out this analysis in a systematic way. Clarifying the need and the expectations of an analysis tool, *E-Ma-Gen3* was conceived in a three-step process. First, existing analysis tools used in other areas were identified. *E-Ma-Gen3* is inspired by the Zachmam Framework™ [123, 124] for Enterprise Architecture and the Unified Software Development Process [50]. These tools provide schemes that enable to examine the properties of the subject of analysis, however these tools are general with a broad applicability that do not allow to highlight specific differences among different *SO* systems. We found that several ideas used by these tools were applicable to the *SO* domain. Particularly, we reuse the concept of "analysis tool" and the "key question" approach (E.g. who, what, where, why, etc.) for

identifying relevant elements of analysis from Zachmam Framework™ and the matrix-based schema and view-based approach from both mentioned references.

The second step, was to identify the gaps required to make the tool more specific to the *SO* domain. It was evident that *SO* systems have properties at different planes, not only cyber digital. So we incorporated a view for the different operation planes e.g. physical, digital or social. On the other side, since *SOs* are active entities we wanted to highlight and differentiate their dynamic nature from other static properties. This way, we identify a view with the a set of fundamental processes the *SOs* might carry out. These processes consume and produce elements from the key areas identified and shape the behaviour of the *SO*. The third and final step was to use the initial version of *E-Ma-Gen3* to perform a first analysis of the *SO* system vision as presented in this chapter and then feedback it with improvements.

The main limitation of *E-Ma-Gen3* is that it is potentially incomplete. First, because it emerged as a tool to facilitate the analysis of *SO* systems rather than with the goal of creating the most complete and best tool for this analysis. Therefore, in this work, *E-Ma-Gen3* is not the aim but the means. Second, because it is part of an improvement cycle where the more *E-Ma-Gen3* is used, the more it is refined, however *E-Ma-Gen3* has been only used in the scope of this work. Finally, because it was conceived with well known existing references in mind (Zachman™and UP tools), the outcome is a tool that share some criteria and structure with its references.

Figure 3.1: *E-Ma-Gen3*: Framework for Analysis of *SO* Systems



The following section present the the characterisation of the *SO* using the framework for illustrating its key elements and concepts. Particular properties of the collective of *SOs* are highlighted when relevant for the analysis. Both a single *SO* and a collection of them might behave as autonomous systems. the analysis tool presented can be used at both levels. The five areas identified are characteristic of both the individual *SO* and the collective of *SOs*. The processes of exploitation, management and generation can be carried out by each *SO* or collectively by different *SOs*. However, there are differences in the challenges that each system faces and that must be considered when engineering *SO*'s software systems.

3.6 SO Analysis using *E-Ma-Gen3*

Since *SOs* exhibit autonomous behaviour, we can analyse them using the tool proposed in section 3.5. Therefore the following sections cover analysis and description of *SO*'s features in relation to the relevant areas, processes and planes.

3.6.1 Planes and Scope

The *SOs* are analysed from three planes: *Cyber/digital*, *physical* and *social*. Every plane highlights a set of relevant properties for the *SO*, its environment and any entity within it. What makes an object smart is its existence in the Cyber/digital plane. The conjunction of elements from the Cyber and physical planes makes the *SO* a unique active entity. These elements are also enablers of the social plane. This one emphasises the *SO* as part of the collective, where other subjects habit, namely other *SOs* or software systems and that all together interact with human individuals and collectives of them. The key factors for each dimension are analysed within each view and process.
The scope of the *SOs*, within each plane, can be clarified by comparing it with other systems:

- For traditional information systems the relevant structure, relationships and resources are part of the Cyber/digital plane. These exploit, manage and even generate knowledge from the physical and social plane but are not able to exhibit behaviour on the physical and social spaces.
- A social bot is defined as "a computer algorithm" [29], it can be seen as a system that permeates the human social networks acting as humans and trying to influence them. Social bots exhibit a behaviour in human social networks but are not able nor concerned to interact within physical spaces.

- Robots are highly active in the physical space. Autonomous motion is one of the robot's key characteristics that distinguish them from the *SOs*.

- Software agents are only active in the Cyber/digital space. They are not concerned with physical properties nor constrained by the physical laws.

3.6.2 Knowledge

The individual *SO* or collective system's knowledge of interest involves elements of itself and its context from the three defined planes. The knowledge about the context can be described using the "five W's" context definition proposed in [2]. The knowledge about itself gives the basis for carrying out the system's fundamental processes along the elements under the areas and planes analysed. We describe key knowledge elements below.

- ***Purpose***: The system goals and motivation. *Core* goals are related to specific system's Cyber-physical characteristics. These are more of the interest of the end-users of the *SO* or the collective. *Support* goals are related to *SO*'s autonomic functions such as to optimise the use of a resources, install and/or configure new components i.e. hardware and software. These are common to multiple *SOs* and domains, being mainly the interest of *SO* administrators.

- ***Properties***: The relevant fixed or dynamic characteristics of itself, other key subjects, the environment, the resources it needs, its components and also its behaviours, functionalities, relationships, among others.

- ***Subjects***: The human individuals, exogenous systems, their collectives and their relationships which are relevant for the *SO* in the scope of each plane. ***Exogenous systems*** are those that neither belong to the *SO* under analysis, nor this is part of them.

- ***Location***: This includes addresses, coordinates or contact details of other *SOs*, systems and human individuals/collectives, resources, components of its structure and its environment.

- ***Time***: References to standard time notations (E.g. UTC, BST) or relative to events or situations (E.g. after X, before Y).

- ***Plans***: Details of how to achieve its purpose considering the existing knowledge.

We illustrate the key knowledge elements with an example. A smart desk lamp (SDL) has the core purpose of lighting when an user is sitting at the desk where SDL is placed. It also has a support goal of monitoring its light bulb lifetime and notify an administrator when this resource gets exhausted. For operation, SDL needs a workable representations of these goals and the following:

- The beliefs about itself, the environment, the collective of *SOs* it belongs to and the relations with them. The beliefs about itself include, for example, the different states of light that can be achieved i.e. bright light, soft light or no light, in addition to the actions that will lead to each one.

- The physical properties of the light bulb —E.g.spiral, LED, white, 12W.—, the place —home or office— where SDL is placed, the lighting conditions of that place and the current time.

- The users to which SDL should respond might include family members except housekeepers or workers in an office except janitors. SDL requires their preferences, usage profiles and their family or workplace relationships.

- For every goal, SDLs needs a plan with event-bounded tasks triggered by either exhausting the light bulb or reaching a lifetime threshold.

- For a human administrator, it will need preferred contact channel —E.g. mobile phone— and the details.

- As SDL is a constrained *SO*, part of the plan for sending notification might require of other *SO* that is able to send and/or display text messages, so SDL will need to know which *SO* can do this part and its URL —E.g. IP address—.

3.6.3 Behaviour

This area considers the most distinctive elements of an *SO*. The *SO*'s capabilities are a function of the elements considered in other areas and represent what the *SO* does as an active entity. Capabilities deliver the functionality of the *SO* and shape its behaviour, these are the mechanisms through which the processes are implemented for the *SO*.

Capabilities are based on the existence of both physical and digital features (i.e. Hardware and Software), but they can be achieved with different configurations. E.g. *SOs* can harvest energy from radio waves or from an *ad hoc* battery, similarly networking

capabilities can be achieved either with wireless or wired interfaces.

Capabilities can be ***Hardware-driven*** or ***Software-driven***. The former ones refer to those capabilities achieved, mainly, by modifying the hardware structure of the *SO*, Software changes are insufficient to reach it. *SOs* can only evolve their Hardware-driven capabilities by incorporating new or improved hardware components to their structure, e.g. adding new sensors that were not originally present in the object. The former ones, refer to those capabilities the *SOs* develop by deploying software routines, models, components or services with minimal (if any) changes in the Hardware structure. For example, a Software-driven capability enables the *SO* to classify the data it gathers and generate further knowledge by using clustering algorithms e.g. k-Means. Once this capability is installed in the *SO* it evolves its original capability offer.

Capabilities can also be ***simple*** i.e. atomic that do not require any other capability to exist; or these can be ***derived*** or complex i.e. these rely on the existence on other atomic or complex capabilities.

Core Capabilities

Core capabilities constitute the most fundamental nature of an *SO*, sine qua non an object cannot be considered smart. These capabilities are atomic, the simplest abilities an *SO* can have. They are based on three of the characteristics identified in [116] for intelligent products, but we extended each one and included a fourth characteristic:

- *Digital Identification*
  It enables information access and object presence in a digital context. This capability requires the existence of an unique and immutable identity but, beyond, it refers to the ability of the object to identify itself to other objects, systems and even humans.

- *Retention*
  It refers to the ability of an object to store information about itself or the environment, minimally its identity. It relies on the existence of a local or remote memory that in more complex cases can be a large repository.

- *Communication*
  It is essential to interchange information with other objects or users. In the simplest *SOs* it is a basic point-to-point mechanism with numerous restrictions –

E.g. any object with a RFID/NFC (passive) tags. At this basic level of communication, objects require others to access Internet.

- *Energy-harvesting*
  As dynamic entities, *SOs* require energy to carry out the processes and tasks they are intended for. This is the the ability to gather the demanded energy either from external sources or by generating it autonomously. Usually, the complexity of the *SO* tasks is proportional to the energy consumption. Therefore, the more energy the objects can get, the more complex the capabilities they can have.

Enhanced Capabilities

Besides the core capabilities, every *SO* might have a set of capabilities derived from the core ones. It is unrealistic to identify all the possible enhanced capabilities, but it is possible to provide a non-exhaustive group of categories that helps to identify the capabilities the *SOs* might have. In some cases, constraints related to the object's purpose, lifetime, design, or just technical or financial matters might be sufficient reason to develop just the core capabilities and a minimum of the enhanced capabilities. From the identified categories, there are two particular ***Hardware-Driven*** capabilities that are the base for most of the other capabilities:

- *Processing*
  It refers to the ability of executing fixed or adjustable instructions and tasks and compute in the background as the object meets its purpose. It relies on the existence of, at least, an attached processing unit such as an embedded controller or system-on-chip (SoC), although some objects can leverage resources by taking advantage of distributed or cloud processing. This capability can broadly vary from one object to other, considering multiple hardware architectures and configurations. Generally, the processing capabilities are required to develop further capabilities.

- *Networking*
  This is the evolution of the communication capabilities with the same central purpose, but involving more complex functions. It implies existence of network adaptors, support to a protocol stack – like network layer protocols as in the OSI Model –, the ability to join a variety of system and object networks and support to multiple communication patterns as the ones introduced by [111] (one-to-one, one-to-many and many-to-one). *SOs* can have intra-networking and/or

Figure 3.2: Smart Objects Capabilities

inter-networking capabilities, for the latter case, it is also required to ensure protection of the object in open environments.

The remainder categories refer to four group of factors:

A *Endogenous Factors*

These allow objects to discover their own features, status, possible changes and issues. Including also the capability to adapt and use information to manage the object's own life cycle and trigger healing mechanisms.

- *Logging*

  These refer to an object's ability of registering events about itself or the environment. In order to log these events, the *SO* has attached storage or connects to remote repositories through its networking capability.

- *Self-Awareness*

  It refers to the capability of an object to know its own status and structure as well as any change on it and its history.

- *Self-management*

  It goes beyond self-awareness and includes the development of abilities to

use the data gathered in order to manage the object's own life cycle including services, response to incidents, problems, maintenance and self-repair. Concrete actions for carrying out the fundamental process of management are encapsulated in this capability.

B *Environment Factors*

These are focused in objects to obtain and improve knowledge as well as discover and manage the environment – both physical and digital – in which it is placed. These include awareness of nearby things, establishing different relations and inducing desired behaviours.

- *Sensing & Actuating*
  According to the purpose, it is common that *SOs* have one or multiple sensors gathering live information from the environment (E.g. home, human body, etc.) or the objects own structure. Although sometimes it is seen as a fundamental characteristic, with the existence on multiple on-line data repositories being permanently updated, *SOs* can still be smart without actually sensing. Actuating refers to the ability of provoking a change either on the environment or on other objects. Usually actuating and sensing are linked, although many objects can have either one or the other.
- *Environment-Awareness*
  These are part of the context-awareness capabilities. These refer in particular to the ability of gathering information from the environment and the surrounding objects in order to improve the user-experience for example by adjusting the object's behaviour. This ability goes beyond having a few sensors; it involves knowledge of environment conditions (E.g. temperature, noise, etc.), locations [99] (E.g. relative and absolute), the present infrastructure and platform, and available services and objects, among others.
- *Social-Readiness*
  This is related to services that enable the object to exhibit social behaviour. For example, joining object social networks and generating and interchanging information in order to meet its purpose and improving services and functionalities offered to the user. Objects are able to relay on and generate social interactions with each other.

C *Human Factors*

Although fulfilling the user expectations is the purpose for what the object was

built and hence it is the driver of any group of capabilities, this group focuses on features and services that improve the interaction with human individuals. In addition, this group considers one of the main concerns of people using *SOs*: trust.

- *Shielding*
  These comprise the services an object offers to preserve the critical characteristics of the information it deals with. These characteristics include availability, accuracy, authenticity, confidentiality – and privacy –, integrity, utility and possession as described in [115]. Regardless of the security mechanisms in networks and platforms with which the object interacts, the object itself is able to provide protection against any threat and thus enhancing trust in users.
- *Human-awareness*
  It is also part of the general context awareness, but it is focused on the services related to gather information from the humans that interact with the object. It includes habits, emotional state, social interaction, spontaneous activity, among others [99]. Since this information is sensitive, shielding becomes a pre-requisite to this capability. It also includes services to improve interaction with human users such as friendly and customised user interfaces. SOs offering direct interfaces to human users without requiring others, are more complex than those which do not.

D *Engineering Factors*
These are related to how the object behaviour is obtained, i.e. how engineers interact with the object in order to induce the desired functionality. In some cases operations are based in detailed pre-programmed instructions, in others, objects have partial control of some activities and in others, advanced objects control every aspect of operation simplifying *SO* creation and management.

- *Programmability*
  It refers to the ability of objects to be programmed. Programming can be fixed (single-time) or dynamic (many-times, upgradable) and can use one or several models (structured, object-oriented, aspect-oriented, etc.). It is highly related to the representation design dimension as proposed in [69]. Objects with this capability require detailed instructions from a programmer to be able to accomplish their tasks.

- *Rule-adaptation*
  It is the ability of the object to modify its operation based on a predefined set of rules in reaction to data sensed from the environment. Engineers can define degrees of freedom for the object and when the conditions are met, the object launches a set of pre-programmed tasks.
- *Goal-Orientation*
  It is an object's ability to act based on defined high-level objectives. Engineers set the objectives and the object is then able to reason and generate the best plan to achieve the objectives. These plans can be dynamically generated based on atomic tasks and considering design restrictions and policies. This behaviour gives more autonomy to the object and enables the existence of other derived capabilities.

The *SOs* compounding a collective might have different enhanced capabilities. These capabilities are then the realisation of the *SO*'s heterogeneity and are a key differential factor among *SOs*. Within a collective, there is a ***Capability Density***, indicating that some capabilities might be common within multiple *SOs* whereas other might be scarce and only offered by a few *SOs*.

3.6.4 Resources

Resources are the physical, Cyber/digital or social elements required by the *SO* in order to achieve its purpose. These can be externally or internally sourced. The latter implies the *SO* is able to use its generation processes in order to self-provide the resources it needs. The *SOs* can potentially exploit and manage all the resources they need for operation and also generate Cyber/digital and social resources.

From the Cyber/digital plane, data is the key resource the *SO* creates, processes, stores, deletes and updates during operation. The data generated by the *SO* can be simple, coming from the raw observations the *SO* makes from the environment, the human individual, their activity and also from other *SOs* and systems. This data can also be complex, which is a result of processing simple data.

From the social plane, the *SO*'s reputation is a resource created, which is based on *SO*'s interactions with human users and other *SOs* and systems. This reputation can be modified either positively or negatively according to the existing rules in the *SO* system. Every *SO* system is also a society that determines the rules for modifying the

reputation, E.g. *SOs* that respond effectively to requests of cooperation from other *SOs* have a strong positive reputation in contrast to those that do not, which have a strong negative reputation and with a wide spectrum in-between.

Physical resources such as hardware components —E.g. a storage unit—, consumables —E.g. paper, cartridges, supplies—, are managed by the *SO*. We distinguish two types: ***Transient*** resources, whose usage pattern is highly volatile, these can be used and recover their initial capacity after usage very quickly E.g. memory; and ***Persistent*** resources, that have a more steady usage pattern, requiring generally a longer process to restore their initial capacity. E.g. the battery. The *SO* controls the usage levels and triggers refill processes that in most of the cases are externally sourced. Generation of physical resources for self-consumption of the *SOs* is rare, with the only exception of energy when the suitable physical structure, resources and capabilities are in place in the *SO*. In most of the cases, physical resources are externally sourced.

3.6.5 Relationships

Many kind of relationships can be established between the *SO*, the collectives of *SOs* and the different subjects of analysis in the *IoT* domain. We discuss some of the key ones:

- *Functional relationships*
  These are the relationships of human individuals who **use/consume** the functionality provided by an *SO* or a collective. Likewise, it is possible to establish these relations between the *SOs*—consumer— and other systems —provider—. In this case, the consumer pursues a goal that might not be shared by the provider, even the provider might not have a goal-directed behaviour. This is a transient relation where the provider only carries out the scoped functionality without awareness of the broader goal being pursued. In this case, the consumer is responsible of the workflow for achieving a goal, it is delegating a concrete task to other *SO* or system.

- *Cooperation relationships*
  This case is well known from agent literature. It implies that cooperating subjects have goal-directed behaviour, they share common goals and are part of the same collective. The two main types of cooperation are knowledge-based and activity-based. Knowledge-base involves sharing beliefs required for the operation of

each *SO*. Activity-based cooperation implies that *SOs* carry out activities in order to achieve the common goal.

- *Social relationships*
  There is a variety of social relationships possible between *SO*-human individuals and *SO*-*SO*. Authors of [11] and [68] identified some of them. E.g. friendship, co-location and ownership. In the context of collective of *SOs*, social relations provide a structure for the whole collective and a position for each individual *SO*. We observe social relations as durable relations rather than short-lived interactions.

- *Composition relationships*
  One *SO* might be **composed** from another system. E.g. platform, operating system or middleware. This is a permanent relation where the resulting composition is the *SO*'s software system.

- *Topological relationships*
  These are spatial relations of one *SO* or a collective, not only with other subjects but also with its physical environment. Some examples are: disjoint, in, touch, cover and overlap relations analysed by Clementini [24].

Since the ultimate purpose of the *SO* is achieved through delivering its functionality. Social and composition relationships only make sense when these are exploited as a way to establish a functional relationship. On the other side, functional, cooperation and composition relationships are the origin of **dependencies** between the subjects at each endpoint using/consuming, receiving the cooperation from or composed from another.

3.6.6 Structure

Relevant structures for the study of the *SOs* are identified for the planes of analysis.

- *Infrastructure*
  This is the specific mix of hardware, an operating system and the middleware over which a particular *SO* is built. *SOs* usually have a fixed and constrained physical infrastructure.

- *Ad hoc Hardware architecture*
  This comprises the specific hardware components, namely, sensors and actuators.

- *Software architecture*
  This includes the software components that are deployed on top of the infrastructure and hardware architecture and are specific for the *SO*'s purpose. Software components that deliver functionality that is common to multiple applications domains are candidate to be moved to the infrastructure of the system.

- *Social structure*
  Characterisation of the social structure is relevant for the collective of *SOs*. The social structure is built from continuous interaction between related *SOs*. These interactions make links/relations between *SOs* weaker or stronger. The *SO*'s particular characteristics, the relations they have with others and its behaviour within the collective make everyone to acquire a particular position in the social structure. Being aware of the position every *SO* holds in the collective is useful for determining which *SOs* to work with.

3.6.7 Fundamental Processes

Exploitation

These processes are applied to elements of each area in order to deliver *SO*'s functionality through capabilities. In other words, the use of knowledge, structure, resources and relationships enables the *SO* to operate. Exploitation involves the flow of processes for making decisions using the knowledge available to the *SO* in order to trigger capabilities according to the structure, resources and relations available. The more complete the knowledge and the capabilities to process that knowledge the more complex the decisions the *SO* can make. High level representation of *SO*'s purpose, relevant properties, subjects, etc.; require a complex process of decision-making [18].

Exploitation involves dealing with uncertainty. Knowledge might be incomplete and still decisions are required for the *SO* operation. Well known approaches for dealing with these situations are: ignore, block and generate.

- Ignoring implies that decisions are only based on and fitted to the available knowledge.
- Blocking implies that the *SO* delegates the decision-making to other exogenous subject and it comes to a blocking state, while the others decide.
- Generation implies that *SO* has mechanisms for the knowledge generation process which will lead to the needed knowledge for making the decision.

Management

These processes are in charge of supporting the operation of the *SO*. If provided by the *SO*, these are implemented through capabilities and allow for achievement of the support goals defined. Management involves essentially monitoring, control and configuration of the elements in each area. For example, these processes might be applied to the different sources and repositories of knowledge the *SO* works with or to the data generated from the operation.

Generation

Generation involves processes for production of new elements, adding up to or changing existing elements, according to each area and plane. From the cyber/digital plane, the *SO* generates knowledge, data, new functional relations and behaviour. For capabilities, the *SO* is able to change the pre defined ones in order to adapt or optimise them as way to achieve its defined goals. An *SO* adapts and enhances its structure E.g. by generating new functional components — E.g. by composing from other more fundamental— or by changing the existing ones —or their relationships—. On the other side, it can increase the physical resources and establish new relationships.

Knowledge generation might be achieved by discovery, reasoning or learning. Discovery is to build knowledge from gathering previously unknown information of itself, the environment or other subjects. Reasoning might be inductive or deductive and involves creating knowledge from the existing one. Unlike previous approaches, learning does not depend on a pre defined algorithm and involves different approaches based on the data resources E.g. supervised, unsupervised or deep learning.

Generation of physical resources is very limited in *SOs* and it is only possible if the physical structure is enabled with the required components. One example is energy. Electrical energy required by the operation of the *SO* might be transformed from energy collected from heat, light or motion and requires specific devices in the *SO*'s structure. In most of the cases *SOs* are not able to generate physical resources.

The social plane is useful looking at the collective. The capacity of adaptation of the collective system depends on the *SO*'s individual capabilities and their state. Adaptation is triggered as a resilience response or as an evolution. Unexpected situations trigger the generation of new relations between cooperating *SOs* which leads to a new

structure aiming to keep the behaviour expected by the human end users. On the other side, the collective system is able to evolve by generating new collective behaviours aiming at either optimising operation for achieving existing goals or seeking new defined goals.

3.7 Smart Object's Autonomy

Autonomy is an expected feature of the *SO* itself and the systems made from multiple *SOs*. It is usually defined from the human user point of view, an entity is autonomous if it is able to carry out tasks without human intervention and control. However, rather than an absolute fixed characteristic to every *SOs*, it is a relative feature derived from other fundamental characteristics of the SO.

SO autonomy is a particularisation of the autonomy defined in the context of agent-based computing (AbC). In AbC, the boundaries defined for the planes within which the *SO* operates are not clearly defined. However, these are necessary constraints in the *SO* context that must be considered in the autonomy definition. Therefore we start by defining autonomy in this context.

The definition is built from concepts discussed by various authors [22, 21] in regards agent's autonomy, enriching it with the elements and constraints given by the *SO*'s relevant areas, planes and fundamental processes as presented in section 3.4.1.

Definition 3 *Autonomy is the ability of a Smart Object to independently carry out the processes of exploitation, management and generation of its knowledge, behaviour, relationships, structure and resources, in order to pursuit defined goals and considering the constraints imposed within the planes it operates. The autonomy depends on and is circumscribed to the goals the* SO *is involved on —either by its own or through the collective it belongs to—, its characteristics and the operation planes.*

To build an operational definition of autonomy in the *SO* context, it is necessary to consider autonomy boundaries. Table 3.1 presents the limits of *SO* autonomy in terms of processes, areas and planes, i.e. the maximum level of autonomy expected by an *SO*. We now discuss these limits from every plane of analysis.

At cyber/digital level, the three fundamental processes can be potentially carried out for the relevant elements by any *SO* for a particular goal. Potentially, there is no limit to

Table 3.1: Limits of *SO* Autonomy

	Cyber / Digital					Physical					Social				
	K	**RC**	**B**	**RS**	**S**	**K**	**RC**	**B**	**RS**	**S**	**K**	**RC**	**B**	**RS**	**S**
Generation	✓	✓	✓	✓	✓	✓					✓		✓		
Management	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Exploitation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

K: Knowledge, RC: Resources, B: Behaviour, RS: Relationships, S: Structure

achieve the three processes in every element for every goal, however, from the current state-of-the-art it is rare that *SOs* are autonomous for every single goal they pursuit in each process and element.

From the physical plane, *SOs* might be able of managing and exploiting their physical elements. They can, exploit, monitor and control its given structure, behaviour, resources, relations and even reason and generate knowledge about these. The limit for *SO* autonomy comes from the object's essence. It implies the *SO* is a passive entity depending on other subjects, mainly human individuals, to generate/outsource physical resources, structure, behaviour and relations.

From the social plane, the structure, relations, resources are not an individual construction of the *SO*. The *SOs* depend on other subjects —part of the whole system— to generate these elements. Social relations depend on the endpoints of the relation, they build the social structure and then these two elements clearly represent a limit for an *SO* autonomy. These are a collective construction, E.g. in the case of the reputation, it is positively or negatively affected from the interaction of the *SO* with others. For any social subject it makes no sense to be autonomous in generation of the mentioned elements, as these are only meaningful in the interaction with others.

Inherent heterogeneity of the *SOs* implies there is a wide spectrum of *SO* autonomy, constrained only by the mentioned boundaries. There is a number of dependencies that hamper the potential of the *SO* autonomy, these come from the different relations of the *SO* with other systems. In many cases these are necessary to expand *SOs* behaviour and achieve defined goals. We are **particularly interested in dependencies coming from the cyber/digital and social planes**, some of the key dependencies identified are:

- ***Structural Dependency***
  The *SO*'s software architecture is coupled to a third system, platform or node that

carries out the fundamental processes for the *SO*. The use of this system, platform or node is not optional for the *SO*. The nature of the dependency of the *SO* with the other system is linked to the localisation of the source of the dependency.

  - *Endogenous*: All the services or functions provided by the other system are deployed and operate from within the *SO*. One example is the operating system of an *SO*. This software system is considered a part of the *SO* as a cyber-physical entity. It does not impose additional restrictions nor lack of control on the *SO*, on the contrary it enables *SO* software to control hardware and network platform.
  - *Exogenous*: it imposes constraints to the *SO* as some services or functions are available remotely to the *SO*, either in other *SO*, a heavy gateway/node or a cloud provider. Exogenous dependencies also imply a transfer of control for some part of the processing, data or other required resource. These dependencies represent a serious constraint to the *SO*'s autonomy for various reasons: (1) the unavailability of the endogenous function blocks the *SO*'s basic operation; (2) there is an added risk, given the channels and third-players required for the function to be carried out.

Structural dependencies are usually set by default to the *SO* from the particular software architecture and development approach followed.

- ***Resource Dependency***

  The resources the *SO* needs are sourced, managed, stored and exploited elsewhere. Particularly, in regards data required for operation E.g. readings about the status of *SO*'s physical properties, if the *SO* lacks of its control there is no guarantee it will be timely available and accurate for the *SO* to make decisions.

- ***Knowledge Dependency***

  Under situations of uncertainty, the *SO* requires another subject —E.g. human individual, system or *SO*— to provide a course of action. The *SO* has no knowledge about unexpected situation, does not recognise them and might end up in a blocking status. The *SO* is unable to learn from its experience and reason about unknown scenarios to determine adaptations the plan for achieving the goal.

- ***Cooperating Dependency***

  Cooperation is usually regarded as conflicting to autonomy [6]. Relying on others requires delegating control of part of the goal to them. However, the cooperation

is natural to *SO* systems for users to benefit from interaction of multiple *SO*. It desirable as a way to expand the limits of each *SO*. Without cooperation some goals might not be achieved.

Whereas cooperating dependency is desirable, the other dependencies might be avoided. Structural and resource dependencies are not specific to a particular goal, but to the *SO*'s software architecture. *SO*'s software might be designed in such a way that these dependencies are avoided. Since it is unpractical to provide the *SO* with all the possible scenarios it might face in achieving a goal, the only solution is to ensure mechanism for knowledge generation. These mechanisms might be common to multiple goals and then incorporated as key components of the *SO*'s software architecture.

The remainder of this work will present a proposal for avoiding structural and resource dependencies by providing a flexible and extensible *SO* software architecture. The software architecture incorporates conceptual elements and capabilities as services. Besides, a middleware architecture provides the basis for building *SO*'s software that is common to multiple *IoT* applications.

3.8 Summary

We have presented a thorough analysis of the Smart Object and its characteristics. We have defined Smart Objects as active socio-cyber-physical autonomous systems with an active behaviour and with a number of capabilities. Likewise, we define the collective of *SOs* as a society built from heterogeneous *SOs* around a defined criteria.

We have analysed the concept of autonomy and presented our own definition in the context of *SOs*. We made this definition concrete to the relevant planes where the *SO* operates, differentiating it from the abstract agent's autonomy. We identified the scope for autonomy in *SO* systems as well as four main types of dependencies that hamper it, namely: structural, resource, knowledge and cooperating dependencies.

We have presented our tool for analysis of the autonomous systems. This tool enables to approach the analysis of these systems from three perspectives: fundamental processes, key areas and planes. The tool was successfully used to describe and analyse the *SO*-based systems.

Chapter 4

Role-based Smart Objects (*RbSOs*)

4.1 Introduction

We have seen in the previous chapter that in order to develop the potential of *SO*'s autonomy, a set of fundamental processes must be carried out by the *SO* while avoiding unwanted dependencies. We have also seen that *SO*'s autonomy is not fixed nor absolute, but it is linked to the inherent heterogeneity of the *SOs*.

In this chapter, we present an *IoT* software architecture based on smart objects (*SOs*). This architecture enables to build cyber-physical autonomous systems that carry out the fundamental processes and gives flexibility for achieving autonomy according to the hardware constraints of the *SOs*. We present the concepts and the software architecture for goal-oriented role-based *SOs* that fit together for the development of *IoT* applications. We built over existing concepts and principles from agent and cognitive systems literature [48, 21, 117, 113]. We will present in chapter 5 how the common functionalities of this software architecture, are detailed and offered as part of an embedded middleware architecture.

In the next sections, we identify the specific challenges and requirements addressed by the software architecture, as well as the key contributions. Then we describe the elements of the architecture and how these constitute the *SO* software.

4.2 Challenges and Requirements

We present key requirements for the software architecture and reference authors when these have been identified elsewhere. A software architecture for *SOs* must:

- *Enable engineering of goal-directed SO's software*
  Since the autonomy is scoped within the achievement of goals, it is necessary that *SO*'s software be designed and developed under the goal concept. This enables the *SOs* to exhibit a truly intentional behaviour where there is a clear distinction of the processes carried out for achieving a goal. The *SO* system must be able to be enhanced adding new goals and the supporting plans and routines for achieving those goals.

- *Enable engineering of SO's software with different levels of autonomy*
  Instead of trying to provide a solution for enabling potential autonomy for every single goal, the approach must intend to set the basic entities for the three fundamental process and enable software engineers and administrators to configure the *SO*'s autonomy according to concrete requirements and hardware set up. The approach must allow achievement of goals at individual and collective level, providing mechanism for adaptation of the system of *SOs* according to the available *SOs*.

- *Provide a model for developing autonomous SOs*
  The architecture must define components and relations required for building *SO* software avoiding structural exogenous dependencies. This architecture should avoid assumptions about a controlled environment where the *SOs* are placed, as this is unrealistic.

- *Provide a simple approach for dealing with SO's heterogeneity*
  Different sources of heterogeneity arise from the diversity of *SO*'s and their environments, more relevant sources to be considered are:

  - *SO*'s hardware architectures [98]: To allow development of *SO* applications that can run in multiple hardware platforms without requiring source code changes.
  - Device-dependant *SO*'s features [62, 101]: To enable applications that incorporate and interact with the different kinds of devices that can be attached to an *SO* (E.g. sensors or actuators)

  - Communication protocols [101]: To enable *SOs* to communicate with each others, with applications, systems and devices using different communication protocols.

- *Enable SO cooperation for goal achievement*
  When *SOs* are not capable of carrying out a set of goals, the approach must provide mechanism for seeking and assuring cooperation of other *SOs* from the collective.

4.3 Contributions

The main contribution of this chapter is to provide a software architecture and a set of abstractions for the engineering of software for *SO* systems enabling adaptability and autonomy at individual and collective level. More specifically, we propose:

- A software architecture for *SOs* based on roles, goal-directed scenarios and plans that are ultimately achieved through the orchestration of known services. This proposal combines agent-computing notions of agent, actions, activities, plans, among others; with a service-oriented architecture.

- An strategy for separation of *SO* behaviour from concrete *SOs* through roles and scenarios. Roles providing a basic interface-type[1] list of responsibilities and scenarios providing more detailed behaviour in a workflow fashion.

- The entities for building autonomous agent-based *SOs* that do not depend on endogenous platforms services, such as directories, nor assume the existing of controlled environment, then boosting the autonomy of individual *SOs*.

- A basic and normalised knowledge representation for the different sources of beliefs an *SO* might need to work with.

4.4 The Role-Based *SO* Software Architecture

The software architecture we propose, enables the development of *IoT* applications based on *SOs* that play different roles within a collective of *SOs*. The roles define groups of functionalities and responsibilities the *SOs* are able to take in order to achieve

[1] In our case interfaces can be instantiated. This is similar to the concept of *port* in some ADLs [94]. For example, in SADL, a *port* has a name, a type and is designated for an input and output [81].

individual and cooperative goals. *SOs* are able to act autonomously, only constrained by their physical infrastructures, but lacking of predefined exogenous structural dependencies.

Our approach is based on the idea that as multi-purpose computers are able to run multiple applications based on their hardware resources, *SOs* can play different roles based on the services they can offer. Therefore, conceptually *SOs* are boxes where roles can be either "installed" or "uninstalled" to achieve a particular behaviour in pursuit of some goals. The services are just wrappings of *SO*'s cyber physical capabilities, that enable modular development. Although there are common roles the *SOs* can play, each *SO* has specific roles according to its particular capabilities, hardware platform and purpose.

In order to build *SO* software following this approach, the software architecture defines the overall components of the *RbSOs* software and identifies the relationships between them. We detail the building blocks for *RbSOs* applications which are the abstractions organised in two functional groups: The uncoupled goal-motivated behaviour and the socio-cyber-physical knowledge representation.

4.4.1 Overall approach

The *SO*'s software architecture is organised in two container layers as presented in figure 4.1. In our case the containers meet three purposes: (1) enabling support to heterogeneous *SO* configurations, (2) facilitating easy reuse of functionality among similar configurations and (3) allowing for runtime adaptation of the *SO*'s structure.

Our approach inherits advantages of a layered architectural style namely: flexibility, reduction of coupling and increase of abstraction, among others [58]. The layers, reduce software complexity by using every layer as a client to the layers below and as a server to layers above. Besides the advantages already mentioned, in our case, this style enables breakdown of *SO* software that is related to capabilities and behaviours providing a natural abstraction for building applications, aligned with real-world entities.

The top agent layer includes both an autonomous software agent and a container of roles. We built this layer over the existing agent theory [117, 104]. The agent provides

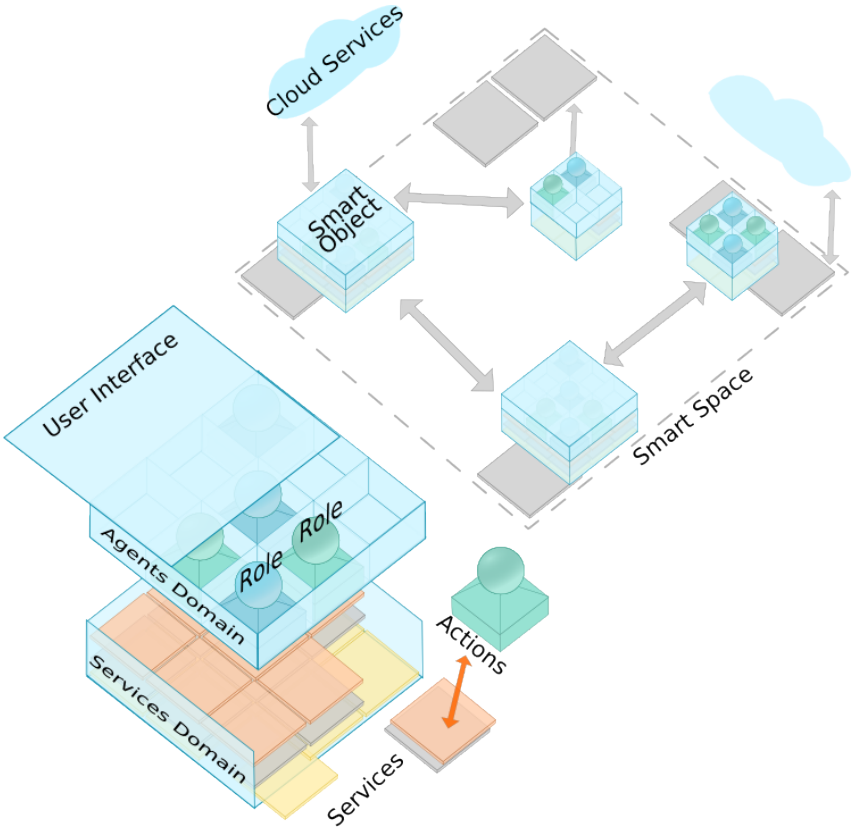


Figure 4.1: Conceptual view of the Software Architecture for Smart Objects

socio-cyber/digital identity for the *SO* and is in charge of the processes of exploitation, management and generation of behaviour, knowledge, resources and relations. The container holds the different responsibilities the *SO* has from a collective's common responsibility definition. These responsibilities are also interfaces of the *SO*'s behaviour expressed as capabilities.

The bottom services layer is a container of capabilities. Capabilities are service components that are developed according to the different combinations of software and hardware infrastructure in place in the *SO*. The container loads and unloads capabilities according to agent layer needs. Capabilities are used in workflows that are orchestrated by the agent layer according to the plans in operation.

SO's software is then a combination of functionalities and abstractions giving support to both layers and also the specific capabilities and roles. Layers functionality which is common to multiple *SOs* is intended to be realised either by a middleware solution (as presented in chapter 5) or incorporated at system level, E.g. as an unikernel solution. Specific development required is aimed to follow a "light programming approach". We call it light programming because from the common reusable functionalities, the development of new applications involves only the agent-based configuration of files and the development of the particular capabilities.

Some known disadvantages of using this layered style include: performance degradation, difficulty to assign functionality to a layer and risk of duplicating services among layers [58]. The performance drawback is more evident when there are multiple layers involved, in our case there are only two layers. There is degradation of performance if we compare our architecture with that of a monolithic *SO* software application tailored to an specific hardware platform. The degradation comes from management services that are required by the containers —in each layer dealing with the software elements i.e. roles and services — that make *SO* software flexible and able to deal with the changing conditions while easing the development effort required by each different *SO* application or *SO* hardware platform.

The performance is a key requirement in some *IoT* scenarios e.g. requiring real-time responses and also considering the constrained resources in the available hardware platforms. However, heterogeneity and volatility are present in any *IoT* environment, solutions designed only for one platform are not able to be used widely as these are coupled

to concrete manufacturers, deploying in a different platform implies building a new application. Besides, performance is closely linked the hardware resources available, with more powerful hardware platforms emerging (see for example the Intel® Joule™ Platform[2]), we consider the trade-off in performance, imposed by the layered approach, is reasonable in order to ensure support to multiple platforms and typical *IoT* dynamics situations.

In the service layer, we use service-oriented components [23] for representing capabilities. This approach enables a clear identification of *SO* functionalities, kept under responsibility of the services layer. What is not manageable by the architecture is the possible duplication of services that could be well supported, as services can be developed separately. This is also a trade-off, we consider justified with the benefit of having the flexibility to support heterogeneous services. A classification and standardisation of interfaces for the services (*SO* capabilities) could help to reduce the potential problem, however this is out of the scope of this work.

In the following sections, we describe the the main elements of our approach.

4.4.2 Uncoupled Goal-motivated Behaviour

The figure 4.2 presents the key entities and their relationships used to represent a Smart Object in the proposed architecture. The Smart Object is a composition of the software agent, the role and capability containers and a particular mix of devices, capabilities and roles. The Smart Object has also a particular set socio-cyber-physical properties which are generally dynamic during operation.

Autonomous Software Agent

The software agent represents uniquely an *SO* within a collective of *SOs*. It uses underling communication protocols supported by the *SO* infrastructure in order to obtain the *SO*'s Id. This agent has not structural dependencies with endogenous entities. Although the end solution requirements define the concrete level of autonomy an *SO* requires, a reasonable expectation — looking to support reference *IoT* requirements e.g. volatility, heterogeneity, etc. — is that these entities should work uncoupled from others (both *SOs* and other type of systems) and be able to carry out a minimal processing by themselves.

[2]https://software.intel.com/en-us/node/721455

The platform services the agent needs for interacting with other agents and to access knowledge base, goals, behaviours and resources are all those located within the *SO* infrastructure. One example is the communication process. Usually, agent platform provides a directory where every agent needs to record the services they might offer. Although this directory might be replicated to reduce centralisation, there is always the need to query this directory in order to communicate with any other agent. This clearly imposes dependency on the *SO* thus constraining its autonomy.
In our approach, there is truly autonomous agent that is supported only by platform services hosted on-object and a P2P communication protocols. This way, even if the *SO* is isolated the agent will be able to perform its basic operation. The agent performs the fundamental processes of the *SO* by picking the available capabilities according to the goal-directed plans being carried out.

Devices

These represent the potential physical capabilities of the *SO*, i.e. the particular on-object and remote sensor and actuators linked to the *SO*. These have id, description, location (on-object or remote), frequency, wrapping service and a set of properties they can either read or change. For example, a light sensor reads the property "Surrounding light", which might be either 0 or 1, according to the physical environment where the *SO* is placed. Likewise, an actuator can be a switch turning on or off a light bulb, then affecting the "Surrounding light" property. The concrete routines performing the sensing and actuating using the devices are wrapped as services. The frequency determines if the wrapping service associated to the device, is in active mode and hence must be triggered according to this frequency value (E.g. in seconds) or, in a passive mode, triggered on-demand by another capability (frequency = none).

Properties of Interest

The properties of interest (PoI) are the narrowed subset of the relevant socio-cyber-physical properties from the environment, *SO*'s resources, the *SO* itself, the human users or other systems —E.g. other *SOs*— of a particular *IoT* scenario. This entity is inspired in the "Feature of Interest" entity included in the OGC's SensorThings data model [86]. The status of these properties is sensed directly through sensing services. Similarly, actuating services are used to directly modify the current values of these properties.

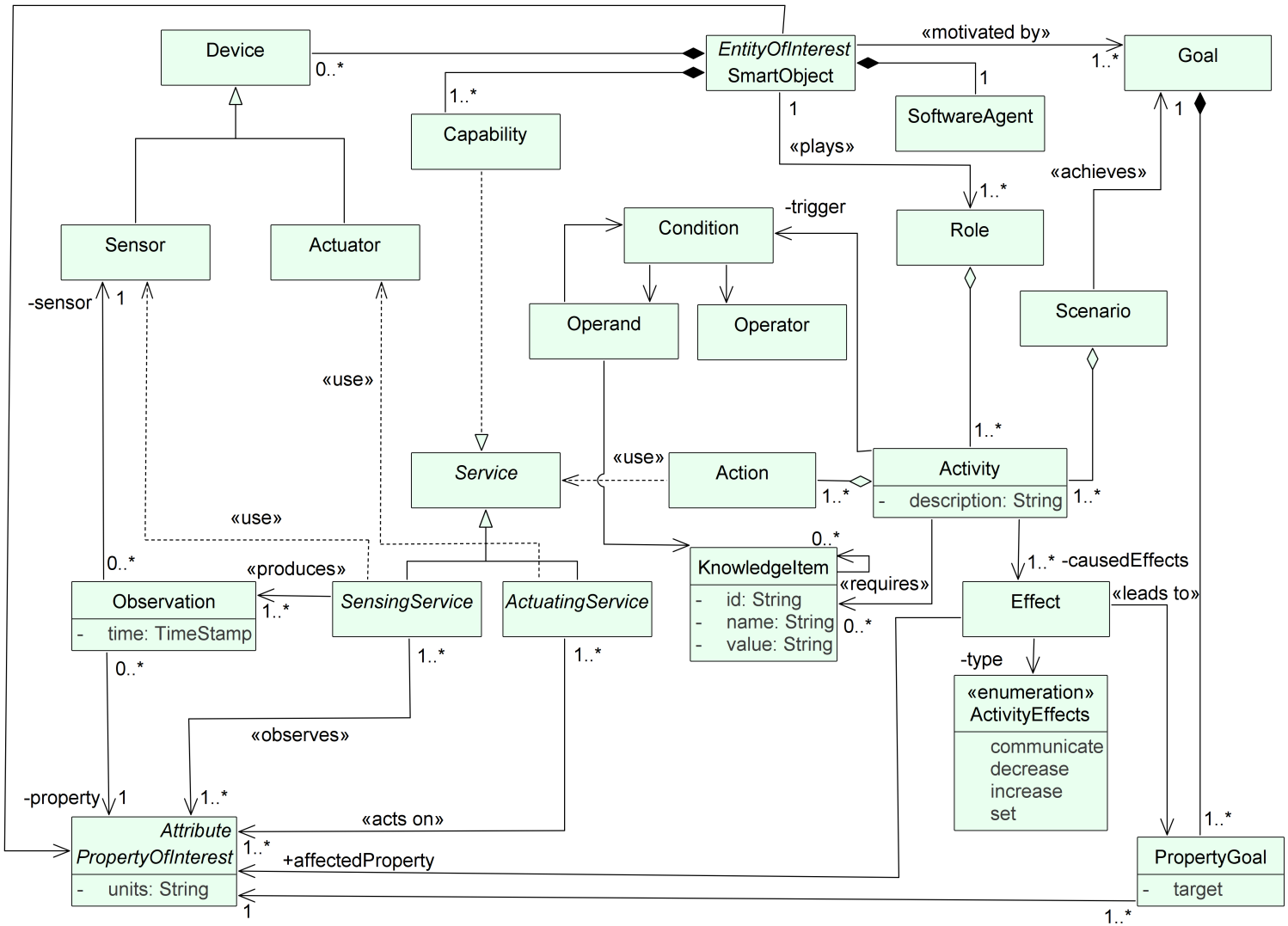


Figure 4.2: Uncoupled Goal-motivated Behaviour

Since different entities of the *IoT* scenario might have common properties, each PoI has a scope, which holds a unique reference to the entity to which the property refers to. Specific observations are transformed into beliefs that are then stored in the Knowledge Base. Properties of interest are dynamic, their changes represent different states of the entities of interest. These are then the basis for defining current and target states along the individual and collective systems. The value domain for each property is defined in the Knowledge Base as a belief and it can be continuous, discrete or categorical.

Goals

The goals define the intentions related to the socio-cyber-physical world which are relevant to the *SO*. We represent goals as target states of the properties of interest previously described. In our case, a goal meets three purposes: 1) It is a conceptual entity (See figure 4.2) that enables the engineering of *IoT* applications, 2) a data unit which is part of the *SO*'s beliefs and 3) a high-level functionality (behaviour) to be achieved by the *SO*. Therefore, goals are stored as part of the *SO*'s knowledge base but also brought to runtime when the *SO* is reasoning and determining which actions to carry out. Goals are conceived to be separated documents, uncoupled from the implementation of any other component of the *SO*'s software (e.g. services, middleware, operating system, etc.). Each goal has the following particular properties:

- Goals are defined with a collection of *SO* or environment properties with specific target states.
- Goals can be time-bounded, it indicates that a constraint after/before a particular date and time can be established for goal to be considered in operation.
- Goals are organised in a hierarchy (See section 2.3.2) including a reference to the goal they belong to.
- Goals have a priority which is established by the administrator according to the nature of the goal. Since the *SO* has a limited processing capability, it will check and try to achieve first the goals with higher priority. This is used to define "selfish"/"unselfish" behaviour in which individual core and supporting goals have higher/lower priority over common collective goals. When a collective goal is prioritised over an individual, the actions of the plan leading to the collective goal are carried out first by the *SO*, this includes the default task of looking for an object to cooperate with when the *SO* does not have the required capability.

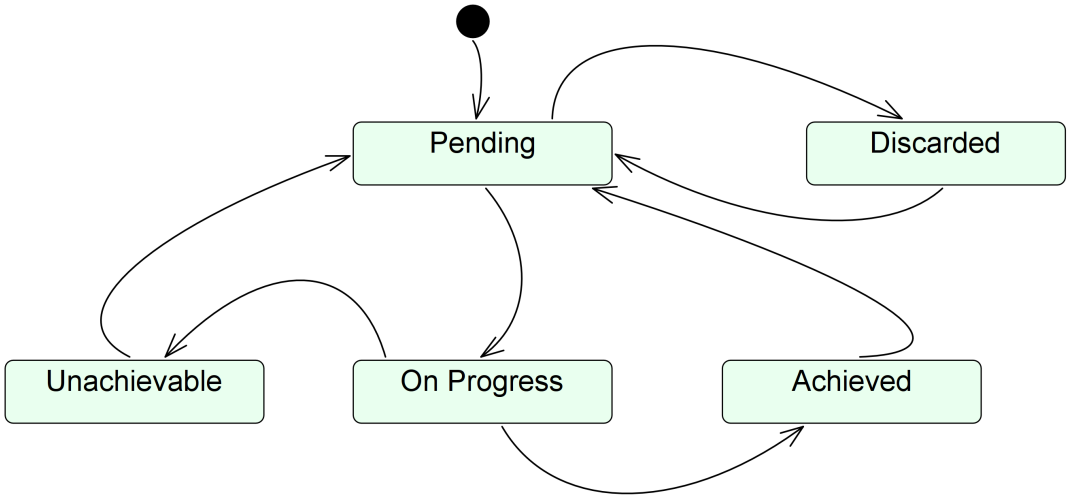


Figure 4.3: *SO*'s Goal States

Goals are processed by the *SO* in order to plan the next activities to carry out given the current *SO* and environment status (See Section 5.5.3). While being processed and according to the state of its properties, we define the goals to be in any of the states depicted in figure 4.3. Goals remain pending or on progress until the target state, of the properties of interest that define it, is reached. The unachievable state is reached when, after a number of attempts of triggering the activities for achieving the goal, the target status is not reached for any of the properties of interest. This could be due to unavailability of the resources or capabilities leading to that state or simply because the goal is time-bounded and the time has expired. After being achieved, goals can transition to the pending state if the target state for the relevant properties of interest has changed.

Activities and Actions

Activities describe a functionality to be carried out by an *SO* through the execution of different known services (capabilities). Activity specifications include the inputs, triggers and outputs in terms of knowledge items (E.g. properties of interest) and the actions. This is a wide definition rather than concrete as every activity can be implemented in multiple ways in different *SOs*. The way activities are defined, enables the specification of scenarios that are uncoupled from the specific capabilities of each *SO*. For example, an scenario can include steps for setting the temperature for a room and

displaying a message to the user, when this is done. In this case, "display message to the user" (including identification of the user) is the activity, that is part of a longer scenario. This activity (and therefore the entire scenario) could be carried out in a different way e.g. using an embedded display or communicating via email or sms to the user. This approach provides flexibility, as scenarios and activities can be defined without knowing the specific underlying services that will execute the operation. Of course, the trade-off is that it adds another level of complexity which also requires maintenance, programming and processing by the *SO*.

The actions are the glue between services (capabilities) and the activities. Besides making use of the concrete services available in the *SO* for an activity, these specify the required arguments and output that are linked to the ones in the activity. The same activity can be implemented in two different *SOs* using different actions (according to their capabilities). At this level, it does not matter how the action is actually implemented, as long as the output is the expected according to the specification.

The output of an activity includes the *effect* it has on a knowledge item. E.g. an activity of an *SO* can have the effect " increase" of the property of interest: "lighting" in a room. Since, ultimately activities are composed by a set of services, one activity can be seen as a simple flow of services, where the agent running it is the single point of control based on the sequence and dependencies defined within the activity. Data flow is also ensured by the definition of the input and output knowledge as pre and pos conditions of each activity.

On the other side, the actions are atomic and specify a particular use of a known services. An action includes the arguments to pass in to the service in terms of knowledge items. An activity can be dependent on updated observations of the properties of interest or on other beliefs from the knowledge base. For example, sending a message to other *SO*, requires the selection and use of a communication service. The asynchronous nature of an agent's message is implemented through services, sending and receiving the message in source and target *SOs* respectively. The message itself in this example, is composed by data gathered through sensing services and also from pre-configured rules in the knowledge base.

Scenarios and Roles

In order to allow for adaptation within the collective of *SOs*, it is necessary to separate the required behaviour for a scenario from the concrete *SOs* that carry out that behaviour. This way, multiple and different *SOs* might have common behaviours that can be offered within the collective. In case of unexpected situations, such as one concrete *SOs* not being available, other *SOs* with a common behaviour can be picked to carry out an activity.

We achieve this through Roles and Scenarios. The Scenario describes the activities to be carried out in order to achieve a particular goal. We use the concept of scenario as an ordered workflow of activities, a collection of dependent functionalities, rather than a sample sequence, as is the common use in requirements engineering. Our scenarios are intended to be defined either by the human users or calculated by the *SO*. Since the activities are by themselves flows of services, the scenario is another level of workflow composition. The scenario works as a template for plans, in fact, the definition of scenarios by human users is similar to defining a library of agent plans, they are loaded in the knowledge base either in design or runtime. If the scenario is not pre-defined, it can be calculated in runtime if the ad hoc reasoning services are available. The reasoning services take the goals to achieve as argument, determine the effect required on the properties of interest to achieve that goal and look for activities producing that effect on the property.

Besides being the entity that ensures predictability of one agent, by scoping the activities and actions it can performs, roles are also a key entity for adaptation. These are fundamental for avoiding structural dependencies at collective level and for the grouping of capabilities. Each role defines an exclusive and unordered list of responsibilities. Roles are kept simple as these are intended to be configured instead of require extensive programming. Roles and *SOs* are uncoupled, the roles played by a particular *SO* can change in runtime. This feature enables the *SO* to adapt its own behaviour to the changing system conditions. Using roles to adapt its own behaviour gives the *SO* the advantage of management of its own structure —based on the underlying services— at broad granularity level, instead of going service by service.

Role responsibilities might be carried out or not, depending on the scenario. A role specification is common for all *SOs* within a particular collective of *SOs*. Therefore, any *SO* able to play an specific role will carry out the same activities. This implies

that, in some scenarios, it is not necessary that a particular *SO* be available, but any *SO* playing the required role, can carry out the activities as long as the role player has the required input knowledge. This characteristic is advantageous as it gives flexibility in regards the particular participants of the scenario. The collective of *SOs* does not depend on a particular *SO* to achieve a goal, improving the autonomy of the collective system.

A role is specified by the human administrator of the collective. Every *SO* has an instance of the role specification. Although role specification are intended to be stable, these might change in runtime. The management of these changes can be approached by having an organisational role, as proposed by Colman (Section 2.3.6).

It is also foreseeable that in some scenarios, although more than one *SOs* can play the same role, it is required that a particular *SO* take over one activity, that should be specified as conditions for the activity. Properties of interest, enable a general specification of roles, providing a common behaviour playable for multiple *SOs* in diverse scenarios differing only in the properties checked. For example, the *SO Resource Manager* role includes activities for resources monitoring and notification to the human users (or other *SOs*) about the lack of *SO*'s resources and asking for replacements; or the *SS Resource Manager* role can include the same activities but applied to smart space resources.

Capabilities as Services

For capabilities we use the concept of service-oriented components from Cervantes et al. [23]. *SOs* wrap capabilities as service-oriented component (we refer to it as service component) covering concrete and simple functionality. Adding new capabilities becomes a process of deploying, on-object, new service components that wrap the operations supporting the capability. As a result, the *SO* becomes compact and able, not only to make its own decisions, but also to carry out concrete tasks in line with these decisions.

In our approach, the service interfaces are intended to be generated by the *SO* when loading (see 5.6.1). The actions and activities use these interfaces for defining the *SO* behaviour within a plan thus composing more complex services for achieving the defined goals, regardless of the concrete implementations. In other words, the implementations are decoupled from the overall *SO* system behaviour. Hence, multiple *SO*

behaviours —following the agent-based notions of activity, action and plans— can be programmed without changing implementation services at this level. Likewise the same services can be easily deployed, in several *SOs* with the right hardware configuration, by copying the service implementation.

Another advantage of this approach is that capabilities are isolated. In the case of an unexpected situation, for example a sensor failure, only the capabilities using that device are affected, being possible to disabled them and even update the roles played by the *SO*, without changes required in the base source code of the *SO* software. In runtime, another *SO* offering the unavailable capability can be asked by the *SO* to run the required functionality.

Each *SO* has a particular mix of common and specific capabilities that are used with the arguments specified in the actions. Capabilities are intended to be based on Support Facilities (Section 5.7). The more the hardware resources in the *SO*, the greater the quantity and complexity of the services. Common capabilities are based on services covering the typical functionality performed by the *SO*, we describe some of them:

- Communication Services
  These services include operations for the configuration of network and human interfaces and the processing required for receiving or sending messages. Network-aimed services receive as arguments: a transmitter, the message type, the message contents and a receiver; and using the supported protocols, they build (disassemble) the message and carry out the transmission (reception) using the configured interfaces. These services ensure at least support to bidirectional communication with other *SOs* or remote systems. Human-aimed communication services require additional processing for delivery and recognition of the message, E.g. speech synthesizer/recogniser processing. Communication services enable indirectly sensing and acting, which is enough in certain *IoT* scenarios.

- Decision-making (Reasoning) Services
  Multiple approaches exist for reasoning (E.g. theoretical, rule-based, ontology-based, probabilistic, etc) and so multiple services can perform different kinds of reasoning based on user-defined rules, preferences, data sensed or particular domain-specific knowledge models. The service approach allows for definition of basic reasoning services giving support to a minimal local decision-making,

that can be extended through these ad hoc services that can be either local or remote.

- Actuating Services
  These services include functionality for modifying the status of (resources and properties of) the *SO*'s and the environment. This functionality depends on the hardware platform and the specific nature of the object. Actuating on other *SOs* or human users is only possible indirectly through communication services. The autonomy an *SO* is increased when the actuating services, provided by this *SO*, enable control of its own properties of interest. E.g Would be desirable that a microwave oven controls the oven temperature based on the conditions of the meal instead of an fixed pre-defined time. For that to be possible, the oven should be capable of either decrease or increase the temperature when required.

- Sensing Services
  Sensing services enable the *SO* to be aware of an updated view of itself and the environment which is needed for a proper decision-making. These services include configuration and operations for data reading from sensors. *SO*'s autonomy is as constrained by the actuating services as it is by the sensing ones. Continuing with the microwave oven example, these sensing services enable the oven to read meal temperature, appearance or smoke presence on-object, and then enable required actions without relying on connection health or remote platforms availability.

- Management Services
  These services extend the basic infrastructure functions offering common operations for different *SOs* that can afford it. Some examples are:

  - *Monitoring Services:* Watch over the operation of the attached devices and hardware platform of both physical and digital resources.
  - *Cost Management Services:* Calculate and monitor cost of services in terms of resources used, transform this information into knowledge items.
  - *Continuity Services:* Manage remote replication and backup of the KB.
  - *Security Services:* Provide operations for ensuring data integrity and proper access to the resources.

4.4.3 *SO*'s Knowledge Representation

The basic intuition of our knowledge representation is to provide a simple solution that can enable on-object storage and reasoning around *preconceived* and *generated* beliefs. As the name stands, *preconceived* are given before the start of the *SO* operation whereas *generated* are created within it, through observation and communication.

In our case, *preconceived* beliefs come from three sources: multi-domain fundamental entities, domain-specific entities and predefined properties of the entities of interest. The multi-domain beliefs are founded on the ontology presented in the previous section, covering the key entities for the motivated behaviour. These beliefs provide the initial *SO* vocabulary and basic relations used to recognise nature and attributes of the entities relevant to the *SO*. This initial set is intended to be expanded for every application with the domain-specific entities required.

The Entity of Interest is used to represent the environment, the human users, other *SOs* and systems that are relevant for *SO* operation. In order to organise and normalise the different properties each of these entities might have, we use the concept of Knowledge Item. The figure 4.4 shows how the knowledge of entities of interest is represented throughout the system.

The Knowledge Item is the entity representing the simplest meaningful pieces of knowledge the system can learn about. It has an unique id, a name/value pair, last usage date, and a collection of attributes also having name/value pairs. Knowledge Items group the different attributes and their values —from the Entities of Interest— used for decision-making. Attributes are the generalisation of characteristics being predefined, communicated or even observed. For characteristics that are highly dynamic, we use more specifically the Property of Interest entity, as described in the previous section.

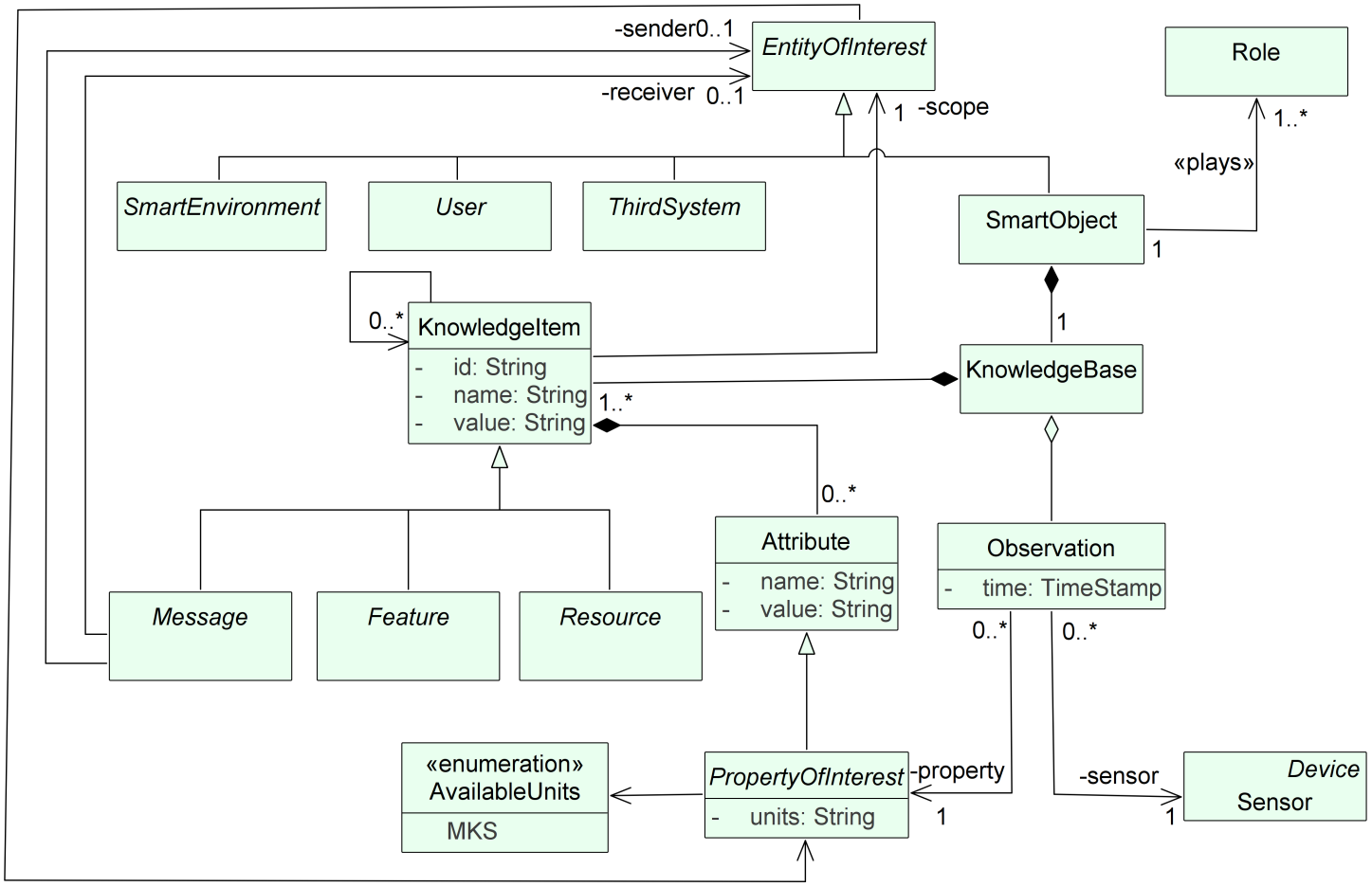


Figure 4.4: *SO*'s Knowledge Representation

We identified three types of Knowledge Items: features, resources and messages. Features and resources are either pre-defined or observed by the *SO* through the available sensors. The difference between each other is that features group attributes that are related directly to the Entity of Interest. E.g. physical characteristics such as location, size, light conditions, among others or cyber/digital features such as software version, available capabilities, number of users, among others.

On the other side, the resource type groups characteristics of the resources related to an Entity of Interest which are not fixed part of them and can be replaced or restored. E.g. battery, light bulbs, storage space, among others. Since we want a simple model, we attach the resources to an Entity of Interest instead of modelling each resource separately.

The messages contain pieces of knowledge communicated by others *SOs* or systems. This entity meets two purposes: (1) enable sharing of beliefs between different *SOs* involved on a scenario and (2) enable to distinguish directly and indirectly observed properties. Messages represent a simplified view of features and resources observed by another system. Messages can be associated to other Knowledge Items grouping features and resources.

Observation represents a raw reading made through the *SO*'s sensors and related to a property of interest. It includes current value of a property, the time the reading was done and the sensor used. This entity is inspired in the "Observation" entity included in the OGC's SensorThings data model [86].

The presented solution enables a basic representation of knowledge relevant to the *SO*. This representation is intended to be used on-object services hence avoiding/reducing the dependency on third systems. In case of requiring more complex knowledge representations, transformations can be addressed in specific reasoning capabilities. These can be deployed on-object —when possible by the available hardware configuration— or consumed as services from specialised remote platforms.

4.5 Summary

We have presented the *RbSO* software architecture. This architecture provides the abstractions needed to build autonomous and adaptable *SO* systems that lack of predefined exogenous dependencies. Smart Objects are seen as containers where multiple

roles and capabilities can be deployed in order to build an *SO*-based software system. Having an autonomous agent backed by on-object platform services boosts the individual *SO* autonomy as there are no dependencies on exogenous directories.

Our approach uses the notions of role, scenario and service-oriented components to decouple behaviour and functionality from every particular *SO*. This way, multiple behaviours can be programmed for an individual *SO*, or a collective, without requiring changes in the services implementation. This approach enables the development of goal-directed *SO* software where the scenarios, for achieving the goals, are orchestrated from known services.

Chapter 5

em4so: A Middleware Architecture for *RbSOs*

5.1 Introduction

In the previous chapter, we have introduced our *RbSO* software architecture and the abstractions for developing autonomous *IoT* applications. We have learned that on top of these, there are some common functions that need to be provided by either a middleware or system level libraries (e.g an unikernel solution).

In this chapter, we present the *em4so* (*Embedded Middleware for Smart Objects*) architecture that covers common functions defined by the *RbSOs* architecture. Engineering of a particular middleware solution must be guided by a set of architectural requirements. We identify in section 5.2 those that we have considered at least partially. In section 5.3, we highlight our contributions in the context of these requirements. Then we give an overview of the architecture to continue explaining its different components. We detail the common interactions between *SOs* made possible through the middleware and particularly the *SO* protocol (section 5.9). We end the chapter by presenting details of the implementation prototype we have developed based on the middleware architecture (section 5.10).

5.2 Research Challenges and Requirements

The middleware architecture must enable the provision of operations that are common to multiple *SO* applications. These can be available as a mix of platform services

available in runtime and as reusable programming routines that access those services. Hence, one first challenge is to determine how to offer the required function. The key is to reduce the programming effort by moving design-time decisions to runtime through configuration of on-object platform services. In addition, for the volatile and heterogeneous nature of *IoT* environments it is important that a middleware architecture gives support to a decentralised decision-making and coordination.

In regards to specific requirements of the *IoT* middleware architectures, these have been identified thoroughly along the literature. We identify here those key requirements that are addressed by the *em4so* middleware architecture. We reference works where these requirements have been identified or compiled elsewhere, if that is the case.

- *Realisation of abstractions*
  Middleware must provide the concrete architecture and a set of reference functionalities using the defined abstractions for enabling development of *SO*-software systems. The focus of this middleware solution is in providing functionality that enables dealing with relative autonomy and adaptation of heterogeneous *SO*'s, not only at individual level, but also at collective level. This way, the software engineer does not need to design and implement new solutions for autonomy and adaptation for every new application.

- *Device Management* [101]
  Offering dynamic discovery and configuration of different sensors, actuators and interfaces (E.g. Network) the *SO* might have.

- *Context Management*[63, 101, 90]
  It refers to processes from information acquisition to reasoning using contextual information including digital, physical and social attributes.

- *Data Management* [101, 37]
  It covers operations to configure generation, store and make available the data generated by the *SO*.

- *Service Management*
  These are related to the catalogue of services offered by an *SO* and its lifecycle. Beyond the service discovery, it also involves dynamically discovering services atop of the particular hardware architecture and making them available/unavailable for an application.

- *Application management*
  Operations to allow an *SO* to dynamically active/deactivate and change configuration of applications running on top of the SO.

- *Social engagement*
  Since the *SO* is also a social entity, middleware must provide functionality to enable *SOs* to engage in social interactions beyond bare communication. Hence, based on the available abstractions, the middleware must enable:

  - Cooperation between two or multiple *SOs*, either performing actions or sharing knowledge.
  - Keep and manage knowledge of previous interactions for decision-making about future interactions.
  - Use knowledge about other *SOs* in a collective to make decisions about individual and common goals.

- *Enable* SO *Evolution*
  This has been identified as a characteristic of *SOs* by Kawsar [62], but not incorporated as a middleware feature. On top of the available abstractions, middleware must support capability upgrades when the *SO*'s hardware configuration allows it.

5.3 Contributions

In this chapter, we address the shortage of practical approaches for decentralisation of autonomic functions (See section 2.3.5) with the following contributions:

- A *em4so* architecture for *SO* software that enables taking advantage of on-object resources for decentralised decision-making at collective level while reduce dependency on exogenous platforms or systems. Thanks to the middleware architecture an *RbSOs* is able to keep working even if there is no connection to other systems or *SOs*. Our focus here is in local (decentralised) on-object processing, in contrast to delegate it to remote platforms.

- A role-based p2p coordination and discovery method for *SO*. This method enables to locate and coordinate with available *SOs* without requiring a remote platform-based address repository. Since a role might encapsulate multiple services, there are less frequent discovery requests compared to do it at service level.

- Definition of an approach for providing autonomic runtime services for the key *SO* processes including lifecycle management, capability discovery and cooperation with other *SOs*.

- Definition of an approach for using information gathered from single communications to drive long-term interactions and adapt to context.

5.4 Middleware Architecture

5.4.1 Design Principles

Middleware is well known as a facilitator of the software development process. Its role, from the software side of the *SO*, is to reduce the development effort. This reduction either comes in the way of providing software libraries with pre-defined solutions to recurrent problems, or it comes as a set o runtime services that transform development effort into configuration and management.

The aim of the *em4so* architecture is to work as reference for building middleware solutions that enable development of *SO* applications, on top of it, with by default support to relative autonomy and adaptation functions. In contrast to leave the *SO* behaviour approach open for the application developer. The approach is to define a minimal set of functionalities that are common to multiple *SOs* and application domains and embed those within each *SO* as the basis for building applications.

The middleware is then deployed in every *SO*. There is no assumption about a common platform to which all *SOs* have access to and provide services to all of them —E.g. directory, semantic reasoning, etc.— Instead, the middleware provides minimal services for each *SO* that enable it to communicate, reason and adapt to changing situations. From the approach used for enabling autonomy and adaptation, the middleware can be:

- *API-dominant*
  It means that *SO* behaviour is enabled mainly through API services, requiring extensive software development effort. E.g. Goal and plan definition involves programming. The *SO* software when deployed only fit for the purposes envisioned during development time. Changes in the goals and plans require software components to be adapted.

- *Configuration-dominant*
  It means that *SO* behaviour can be configured from the middleware services without changes in the software components. E.g. Configuration-dominant goal definition gives users direct access to drive *SO* behaviour by defining new goals that fit the deployed software and hardware architectures.

The Configuration-dominant approach is preferred, as it provides a straightforward support to delegation, through goal definition. It is also more secure as goals and plans do not require changes on core source code and these are constrained by the existing *SO* software, limiting the scope of the modifications.

Enabling Social Features

As an agent itself, the *SO* is situated in an environment, therefore social abilities are another key characteristics of the *SO* which are common to every application domain. Use of the knowledge about social interactions is key for adaptation at collective level but also can be used for *SO*'s individual decisions. Subsuming agent and social theory literature [12], the following have been identified as key *SO* features that can be enabled from middleware services and routines:

- ***Communication***
  Communication is a well known and a common use of the middleware infrastructure. The communication approach must avoid constraints on other *SO* features, particularly autonomy. At application level communication, pure peer-to-peer approach avoids exogenous dependencies which makes it suitable for *SO* communication. Middleware then must provide mechanisms to create an overlay network in order to enable P2P *SO* communication. At lower levels, middleware must enable the developer to deal with heterogeneity of protocols and avoid constraints to particular choices. It must enable utilisation and selection among available choices according to each *SO* hardware resources.

Table 5.1: Infrastructure functional blocks for architecture

Agent infrastructure	
Lifecycle Management	Create, activate and deactivate software agent
Knowledge Management	Create, maintain and store knowledge
Reasoning	Make decisions based on existing knowledge
Perception	Monitor and pro actively sense environment in search for stimulus and changes
Communication	Receive and send messages from other agents
Cyber physical infrastructure	
Device Management	Discover, active, deactivate cyber physical devices
Context Management	Acquisition, modelling and reasoning [90]
Data Management	Sense, store, clean and prepare data of interest
Application Infrastructure	
Service Management	Discover, publish, maintain and select services of interest
Application Management	Create, activate, deactivate supported *SO*-based applications
Autonomy Framework	
Goal Management	Definition, organisation, prioritisation, update and removal of goals
Plan Provision	Definition of strategy and configuration for plan generation E.g. definition and discovery of pre-designed plans within a library

- ***Cooperation***

  Relying on other *SOs* in order achieve a goal, involves delegating control of part of the goal to them. It is not the function of the middleware components to determine the exact balance of cooperation and autonomy, i.e. to what extend, a task is delegated, and the data required to achieve that task, is distributed. This balance depends on every particular use case and application domain. The role of middleware goes until providing services that enable cooperation. Based on communication protocol and services, middleware architecture must provide a mechanism for enabling *SO* cooperation. The cooperation types to support include knowledge-based and activity-based as discussed in section 3.6.5. That means the definition of a coordination approach between *SOs* within a volatile environment is also required. The use of roles as part of the conceptual model enables flexible coordination uncoupled from specific *SOs*.

- ***Collective Adaptation***

  The social context of the *SOs* is dynamic and includes mainly other *SOs* within the environment and the interacting end-users. From an internal view, the *SO* must adapt its operation to the context. For a default adaptation strategy the middleware must include services dealing with cooperation scenarios. The *SOs* are immerse in a environment containing large number of *SOs*, it is key to select the more suited *SOs* to cooperate with. The selection made for one scenario is not static but it should be prone to change and adapt to new conditions as the *SOs* and their properties are dynamic. Selection of the *SOs* can be based in multiple criteria, for example related to hardware resources —E.g. available resources—, performance, location and trustworthiness —E.g. the time *SOs* are known by each other–, among others.

5.4.2 *em4so* Architecture Overview

The purpose of the *em4so* middleware architecture is to provide the common functional components that enable the realisation of the *RbSOs* approach. This way, applications can be designed and built on top of middlewares implementing this architecture. Since the *SO* is a single Cyber physical entity, the architecture covers not only functional/logical aspects but also physical deployment considerations to enable different configurations according to application requirements.

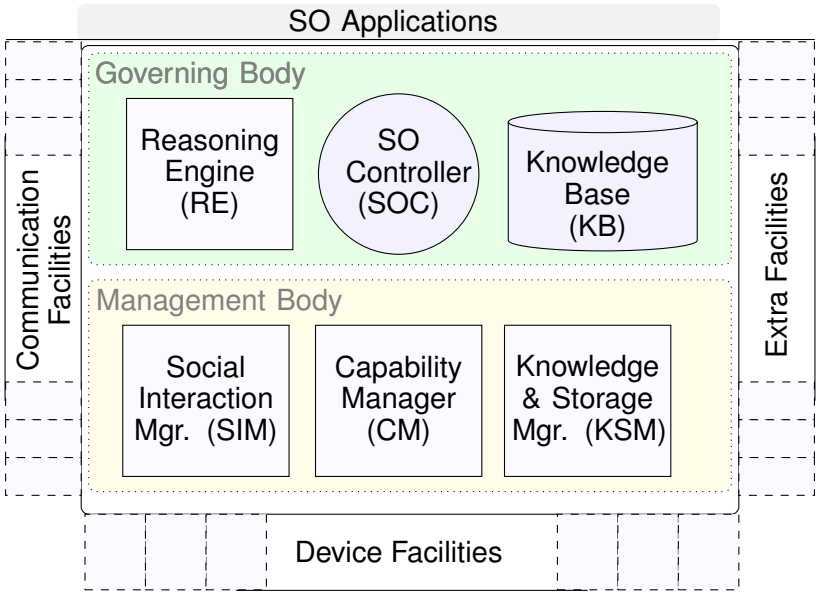
From middleware requirements and building from literature [118, 21, 104, 30], functional blocks satisfied by the infrastructural component of the middleware are identified in table 5.1 on page 110. The proposed architecture is presented in figure 5.1, this is built from our work presented in [92]. The architecture presents the central components arranged in two levels and a set of supporting facilities. This structure is inspired on the enterprise organisations. The idea is to distinguish the functions related to the regular operation of the *SO* or particular parts of it —Management Body— from those concerned with the overall achievement of the *SO* goals —Governing Body—. These functions are fundamental to any *SO* and so are common along multiple application domains. How these functions are implemented and deployed has impact on the overall autonomy of the *SO* i.e. it affects all defined goals. In addition, the architecture groups those functions that might vary from one *SO* to other and that require ad hoc programming efforts —Support Facilities—. How these functions are implemented and deployed has impact on the autonomy relative to particular goals.

These facilities are not coupled to the a central components of the architecture and have multiple purposes. These can vary and allow for tailoring to specific *SO* hardware platforms and supported communication protocols. Besides, these also enable the evolution of the *SO*'s central infrastructure as facilities can provide extra services for enhancing management and governing functions. Finally, these facilities allow for reusing of existing frameworks and APIs specialised in dealing with particular problems. E.g. hardware manufacturer APIs, data storage or network communication.

The specific middleware solutions, implementing this architecture, must provide runtime services for the two identified bodies and API access for the particular *SO* platform services. E.g. A Capability Manager (CM) runtime service must deal with the lifecycle of the capabilities deployed in an *SO*, whereas the capabilities have to be programmed individually and made visible to the CM through the use of the supplied API. This way, the architecture allows to achieve different deployment configurations by exploiting the resources available on object.

Since the *RbSOs* approach is driven by the agent paradigm and MAS, the most natural way to make it concrete is to rely in an agent platform. An agent platform provides then the basis which enables middleware functional blocks. Rather than imposing dependencies, the agent platform services are deployed locally, and with the *SO* communication based on a p2p protocol, the *SOs* have what they need for its basic operation.

Figure 5.1: Logical View of the *em4so* Architecture



There is a trade-off between the adaptability given by moving decisions from design time to runtime and the easiness of the quality assurance process for the system. In other words, the middleware, built using the proposed reference architecture, enables the development of applications that can change in runtime making it difficult to specify, validate and verify the *SO* systems at individual and collective level. This is a characteristic of adaptable systems because their behaviour is not completely deterministic. This lack of determinism brings risks to the software system as whole, including reliability, security and correctness. For example, it is not possible to predict which *SO*, in particular, will be used to complete a scenario. If a failing *SO* is picked, it might cause a malfunction of the whole collective ending up, in a scenario failure. Besides, an untrustworthy *SO* represents a threat as the capabilities offered can be used to exploit collective vulnerabilities.

One way we mitigate these risks is through the Social Interaction Manager (SIM) and the *SO* protocol. First, the *SO* protocol enables the definition of rules for the formation of collectives, enabling filtering of *SOs* that join a collective (See section 5.9.1). Second, since the lack of hardware resources can be a source of failures in *SOs*, the SIM gathers information about *SO* members of the collective including the *Declared Resources Profile* and the *Current Resource Availability* (See section 5.7.2) for using when selection the *SOs* to collaborate with. Finally, the Extra Facilities (See section

5.7.3), enable the middleware architecture to extend incorporating additional security or reliability monitors. This is not ideal solution for scenarios with strong safety and security requirements, however most smart scenarios in homes, buildings, industries or cities that benefit of the flexibility and adaptability provided by our architecture.

In the following section we provide further description of the *em4so* architecture components.

5.5 Governing Body

This body performs decision-making, defines and controls how the whole *SO* infrastructure, capabilities and applications are managed in order to achieve the defined goals. The focus here is on the processes related to the purpose of the *SO*, i.e. what the *SO* has to carry out. These components ensure that decisions are taken, balancing the different goals that have been set for the *SO*. It also ensures control of the plan lifecycle for achieving the goals, which is the way of controlling the overall application flow.

5.5.1 Smart Object Controller (SOC)

This is the autonomous software agent responsible for keeping the *SO* up and running according to goals specified. For doing so, first of all it gathers information about itself and the environment (context), making it available for the *SO* operation by populating the KB through the KBM. This setup process involves the loading of available goals, the *SO*'s infrastructure, applications and the relevant properties of interest from the *SO* and the environment as well as the available capabilities to sense these. One example of the information the SOC obtains in the setup, with support of the Device Facilities (Section 5.7.2), is the *Declared Resource Profile* (*DRP*). This is a set of calculated values from the *SO*'s hardware configuration that give an indication of the *SO*'s power and are shared with other *SOs* of a collective for the *SO* selection (Section 6.5).

After setup, architectural components and capabilities are in charge of updating status of loaded entities (goals, capabilities, properties, etc.) through KBM. SOC then queries for the updated status through KBM. Based on the goal status, it triggers reasoning process through the RE and determines which actions to carry out. This is a cycle as the one depicted in Algorithm 1 which is an adaptation of the abstract agent loop proposed in [104]. Actions to perform are not only reactive to changes in the *SO*

context but also pro active based on the existing knowledge.

The SOC monitors Management Body, triggering particular operations on each component based on the goals and the decisions made through the RE. This component makes possible the goal-driven behaviour of the *SO*, which is a condition for the relative autonomy. The design of this component enables goals to be defined as uncoupled configuration documents that might change during runtime. The SOC monitors these changes and trigger the required reasoning process to deal with these.

When working towards a goal, the SOC uses the RE to determine the plan and activities to carry out for that goal. The SOC controls the execution flow for each plan and then, it is in charge of using each specific service linked to the actions of each activity. The SOC follows the plan and activity workflows, it uses the CM to localise on-demand the service associated to each action. In order to process each action, the SOC checks that preconditions defined for each activity, through the activity definition, are met.

Each workflow is intended to be implemented as separated processes/threads, in that way, each process/thread is blocked until it receives the references of the services it requires, but it does not block other workflows. After a service associated to an action is triggered and an activity is completed, the SOC checks the preconditions of the next activity to keep working until completing the plan. When the service output involves changes in the states of the attributes of entities of interest, the KSM is triggered in order to transform those in beliefs to update in the KB.

```
   input: A boolean for periodic checks of SO's structure: setupRequired
1  /* goals, activities, roles, capabilities and
      devices are part of the KB available at global
      scope                                          */
2  if setupRequired then
3  |  checkDeviceChanges(devices);
4  |  checkCapabilityChanges(capabilities);
5  |  performKBMaintenance();
6  end
7  pendingGoals ← traverseGoalHierarchy();
8  for goal in pendingGoals do
9  |  observations ← observe(propertiesOfInterest);
10 |  updateKnowledgeBase(observations);
11 |  activities ← reason(goal); // goal conditions satisfied
   |   & plan (list of activities)
12 |  for activity in activities do
13 |  |  triggerActivity(activity) ; // include the activity
   |  |   in the set of pendingActivities to be executed.
14 |  end
15 |  updateGoalState(goal);
16 end
```

Algorithm 1: High-level *SO* Controller Loop

5.5.2 Knowledge Base (KB)

It is designed to store knowledge, relevant to the *SO* operation, either generated by itself or by the context. Beyond the entities defined in sections 4.4.2 and 4.4.3, no other particular knowledge representations are enforced, leaving that open to middleware implementation. The KB is composed by the set of beliefs of the *SO*. The architecture allows for storing of *preconceived* and *generated* beliefs (Section 4.4.3). The set of *preconceived* beliefs related to the multi-domain entities is based on the ontology presented in section 4.4.2. These are part of the middleware architecture and come pre-stored in the KB. The remaining *preconceived* beliefs are specific for the domain or application and must be loaded, by application programmers, for every particular implementation. These Knowledge Items and attributes include, for example, measure-

ment units, existing human users, types, locations —E.g. living room, meeting room, office, home, park, etc—, their topological, composition and different types of relations, among others.

The KB persistence strategy is particularly important to ensure the *SO* autonomy. KB is designed to be distributed and based on semi-structured data storages. As the KB incorporates the regular observations according to the *SO* applications, the repository might grow very fast. The KB is designed to have a cache repository on-object, this is refreshed and managed by KSM. This cache includes the core multi-domain model (E.g. goals, roles, activities, plans, etc.) and the most recent used Knowledge Items. This way, *SO* will be able to make decisions and operate even in connection-less situations.

Control over KB contents is approached through determination of *access policies* for the KB, mainly applied on the data generated for the properties-of-interest and the offered services (capabilities). In this way, there are three levels of access: *private* —only available for the *SO*—, *public* —available for any object or application— and *specific* —allows to define a set of authorised URL. These policies are considered when the KB is exploited for a particular activity via the KSM.

5.5.3 Reasoning Engine (RE)

It enables the *SO* to develop beyond a purely reactive behaviour. The SOC component requests the RE to performs the decision-making process, choosing activities to carry out and the plans to follow. The RE is designed to provide a basic solution for decision-making on-object. The reasoning is inspired on the means-end reasoning used in the Belief-Desire-Intention model proposed by Georgeff [40] (See section 2.3.1). This reasoning, also known as *practical reasoning*, is based on deciding what *state of affairs* to achieve and how to do it [117].

In our case, the RE determines the next goals to execute by examining the properties of interest being monitored. It receives from the SOC the work to be done in terms of goals with unachieved status. Then, it evaluates goal conditions according to pre-existing and performed observations in order to determine which plan to invoke given the current status. Generic scenarios are pre defined in the KB, these define template plans including: list of activities, conditions to be met for triggering, required knowl-

edge and data to process. The RE chooses the scenario linked to the goal. Once the SOC triggers the scenario, the RE queries the KB to obtain the current beliefs that match the evaluation terms.

The RE allows for querying over pre defined goal plans or for generation of the plan based on available activities. Meta-data for each activity includes post-conditions in terms of *properties of interest* which can be matched to goal's target state —also expressed in *properties of interest*— in order to *backward-chain* the plan generation.

When reasoning for triggering goal plans or determining if conditions of an activity have been met, the existing beliefs might not be enough to determine a resulting action for a given state of a property of interest. The RE calculates new relations based on the existing ones. For example, based on the topological relations the location of one entity of interest can be obtained, even if the location is not recorded as one of the attributes of that entity. Concretely, if an Object A is next to other B which is within the living room, the RE determines that both A and B are within the living room even if there is no relation between the Living Room entity and the Object A entity. An attribute representing the fact that Object A is also within the Living Room entity is then generated trough the KSM for Object A: "within: LivingRoom". Hence, the observations coming from any of these objects can be used for evaluation of conditions for a particular activity or plan.

The on-object RE allows for autonomy from cloud platforms. However, the process is highly constrained by the resources available. A more extensive process would require capacity for both processing and storage of huge amounts of beliefs. For this reason RE also works as a proxy, enabling consumption of local or remote services provided by more powerful and specialised, reasoning platforms (Section 4.4.2). This design enables configuration of different degrees of autonomy from reasoning platforms.

5.6 SO Management Body

The management body is the group of three managers that enable the *SO* operation with different degrees of autonomy. The focus here is on how the *SO* carries out, executes, the decisions made at Governing level. These managers monitor and act, using their facilities, over their elements of interest. E.g. capabilities, knowledge base, or communication protocols. The purpose of this body is to uncouple the key multi-domain *SO* functions from the actual constraints imposed by the particular hardware

and software platforms. E.g. a network interface. In addition, instead of enabling API-level interaction to the programmers, these managers offer runtime services based on the *SO* configuration. These are aimed to ease both programming, as routines are provided by the *em4so* architecture, and management tasks during the *SO* operation. Having this body per se does not implies a particular degree of autonomy relative to a goal. This varies, according to the deployment of the capabilities, the knowledge and communication services.

5.6.1 Capability Manager (CM)

This is an autonomic manager responsible for discovering, on-demand loading/unloading and configuring of the capabilities required for the *SO* activity execution. The capabilities are realised as service components that are developed atop of the available hardware and software platforms. Therefore, the capabilities are not coupled to the middleware architecture itself but are part of the specific programming for each particular *SO*. The capabilities are deployed in the local directory recorded in the *em4so* configuration files.

The CM works at service level and has a double role, one as consumer and the other as provider. As provider, the capability manager detects, at runtime, available capabilities, extracts the meta-data describing them (E.g. contract, data parameters, host/location, category, etc.) and store it in the KB via the KSM. This way it is accessible by the RE when determining how to deal with a particular *SO* activity. It also updates the capability locations when these change if the *SO* is mobile. The capabilities are specific for each *SO* and can be based on hardware components (E.g. sensor, actuator, network interfaces or peripherals), other API-enabled software libraries or ad hoc programmed routines. In order to use a particular capability, this manager dynamically loads execution instances based on the stored meta-data. It is in charge of either the creation of new instances or the allocation of existing stateless objects having the capabilities/services implementation. The specific data to use when invoking the services comes from the KB. It can be either gathered from the *SO* operation, pre-defined by the *SO* Developer or configured by the *SO* (End-user) Manager.

As a consumer, the CM is responsible for allocating the capabilities (services) required by the *SO* for a particular plan. It carries out the search, selection, request and usage of the services required. These services can be hosted locally or in remote/cloud plat-

forms. It allocates the services that best suit the decisions made in the Governing body and binds them making them ready to be used by SOC. The selection of services is part of the adaptation strategy and is presented in section 6.6. If a service is hosted by another *SO*, the request for using that service is managed through the SIM component. The selection of a particular service has impact in the autonomy of the *SO*. The motivation of one service or other comes from the given goals and the current state of the *SO* itself —E.g. resources— and its context. Different configurations of on-object and cloud services are then possible resulting in different degrees of autonomy for specific goals.

5.6.2 Social Interaction Manager (SIM)

The SIM emphasises the existence of the *SO* as one entity within a social context, surrounded by other *SOs*, systems and end-users. This is a runtime service with two purposes. The first is managing the communication of the *SO* and other *SOs*, systems and end-users. The second, is about adapting the *SO* to the social context by managing the roles the *SO* plays within an collective for the existing scenarios.

This component goes beyond establishing communication and transmitting/receiving messages to a particular actor (E.g. *SOs*). Since every *SO* has a reduced *"contact list"* including the known *SOs*, this component triggers the routines to manage that list based on the previous interactions the *SO* has had with those in the list. The *"contact list"* records not only the network location (E.g. IP address) but also a reference to the physical location which is transmitted by the neighbour *SO*. In this way, the SIM cleans the list removing *SOs* that do not meet the criteria which comes from the user preferences matching the actual interactions with every *SO*. For example, if an *SO* is committed to carry out an action within a time window but it did not accomplish it, this could cause SIM to flag this *SO* as candidate to leave the *"contact list"* if a slot is required for a new *SO*.

The SIM also considers long-term interactions instead of single communication acts. It records available information about the *SOs* with which it interacts in order to consider it for decision-making by Governing Body. When queried, *SOs* store key information from each others. The SIM is responsible for triggering update and cleaning routines for the KBM in order to keep KB updated with relevant cooperating *SOs*. This process is progressive and depends on the available connections.

This component encapsulates the heterogeneity of the communication protocols and the participants. The SIM includes a message processor which receives and transmits *SO*-level messages according to the *SO* protocol (Section 5.8). It uses the communication capabilities based on supported protocols to receive/transmit messages from/to senders/receptors. The capability must include the mapping between the *SO* protocol and the particular communication protocol implemented.

Roles played by an *SO* are dynamic during its lifecycle. The SIM ensures consistency of the common roles and scenarios definition and enables or disables specific roles according to decisions made by Governing Body. To ensure consistency, for each collective the *SO* might belong, the SIM has two mutually exclusive modes: *change promoter* or *change adopter*. *Change promoter* is only activated through a special configured role: the *collective organiser*. This role is intended to be initially assigned to the *SO* which is the collective founder, however it might be moved to another *SO* by the *SO* founder administrator. The *SO* playing the *organiser* is the only enabled to propagate changes on the roles and scenarios. This way, conflicts between role/scenario definitions are avoided as there is always a unique valid definition. If the collective founder is not available, the available *SOs* can still keep working with the definitions they have available —local copies—. When joining a collective, every *SO* whose SIM is in *change adopter* mode, overwrites its own role definition (instance) with the one provided by the access *SO*[1]. In addition, on start up and periodically when changes on the roles are received or changes in capabilities are communicated by the CM, the SIM checks that activities included in the role are supported by the *SO* capabilities and that bounded remote services are available. When the role is not fully supported, the SIM disables it.

The SIM enables the *SO* to control the type of interactions it is able to engage in —E.g. the message request is going to receive from other *SOs*– by disabling playing roles. This adaptation can be triggered for example when there is a lack of resources in the *SO*, E.g. battery. The SIM records the *SOs* within the network that are playing a particular role. When disabling roles, to avoid disabling a scarce one, the *SO* Administrator might include a rule for the SIM including a threshold of a minimum roles available to allow disabling a role. The SIM component allows the *SO* for self-configuration of different degrees of autonomy from other *SOs*. These configurations arise from the goals

[1] the *SO* that has served as access point to an entrant *SO* to join the collective

defined and the available role players. More details on the role management, as part of the adaptation strategy, are presented in section 6.7.

5.6.3 KB and Storage Manager (KSM)

It provides access to the KB to the rest of the components. It encloses CRUD methods altering the contents and the structure of the repository hosting the KB. This is the only component accessing the KB, when required by the SOC. The main operations run by the KSM are:

- Transform raw observations into Knowledge Items. The transformation consists of update current value for attributes of the Entities of Interest to which the observation is related to.
- Create new Knowledge Items derived from the RE outputs.
- Ensure a periodic replication and cleanup of the KB based on pre-defined *SO* configuration.
- Monitor use of beliefs to determine candidates to be excluded from the cache

The KSM component can be extended by ad hoc capabilities providing for example remote backup and storage.

5.7 Support Facilities

These are a set of functions provided via API for the development of capabilities. The Management Body is able to discover and handle capabilities as long as these are using the respective facilities.

5.7.1 Communication Facilities (CF)

These provide routines supporting protocols at different levels of the network stack that enable or extend the *RbSOs* communication via the *SO* protocol. The *SO* protocol can rely on other application protocols —E.g. agent-based, service-based, etc.— or in protocols from the network or lower layers of the OSI Model. The agent-protocols are provided by the platform APIs that encapsulate the low-level network communication programming. Likewise, web service protocols enable discovery, publishing and consumption of web services on top of web protocols either RPC-based or REST-based.

Clearly the programming effort required for a particular *SO* application in regards communication capabilities will depend on the level of the facilities available for the development.

5.7.2 Device Facilities (DF)

When capabilities depend on sensor, actuator or other physical/hardware devices they need to be configured and its access enabled for development of ad hoc *SO* capabilities. This is a common component required and proposed in other Smart Object middleware. In our proposal, it provides interfaces for configuration and use of the attached sensors and actuators. Methods for adding, removing, restarting, reading, activating and deactivating these devices are invoked by the SOC. These interfaces are standard, hiding specific functions provided by Hardware manufacturers. This component calls underlying low level libraries (E.g. operating system) to adjust sensor and actuators according to *SOC* instructions. For example, DF provides the methods for obtaining the different properties of the *SO*'s hardware platform. These properties are used to calculate the *Declared Resource Profile* and include maximum memory, CPU cores, CPU cache and CPU clock speed, among others.

5.7.3 Extra Facilities (EF)

These enable the enhancement of the fundamental architectural components. E.g. the RE or the KSM. The EF offer specialised functions for example covering reasoning, autonomic management, repository administration or data backup, among others. These facilities can be light offering only functions to access platforms were the services are hosted E.g. a REST client; or these can be heavy providing the expected functionality to be exploited on-object.

5.8 *SO* Protocol

Middleware includes routines for enabling communication and cooperation following the principles of the relative autonomy. One fundamental function provided by the middleware is to enable the *SOs* to discover other *SOs* they can cooperate with. This is possible thanks to the definition of an *SO* protocol. The protocol is part of the Communication Facilities (Section 5.7.1) provided by the middleware architecture. The *em4so* protocol is a simple message protocol based on the Gnutella protocol [38]. Gnutella is a peer-to-peer file exchange protocol whose approach offers independence of any

central directory —or any robust (super) peer— which makes it suitable for avoiding constraints to the autonomy of the SO.

The *em4so* protocol adapts Gnutella approach to the *SO* context and incorporates additional elements. The peers are *SOs* and instead of addressing file exchange; roles, players and activities are the entities of interest of each peer. The resulting protocol is basic but cover the key interactions between *RbSOs*, enabling not only communication but also cooperation. As Gnutella, the *em4so* is an application level protocol that is aimed to work over internet protocols. Messages are composed by a header and a payload. The header carries the meta-information of message: a message id, the Time-to-live (TTL), hops and a payload descriptor. The novel elements of *em4so* protocol are summarised below.

- *New primitives*
  Four new primitives are incorporated to allow coordination for cooperation based on activities and also for knowledge transfer. Table 5.2 presents the complete set of primitives and its description.

- *PING & PONG payload*
  PING messages include a payload. This is required to describe the entrant *SO* to get access to the overlay network as explained in section 5.9.1 and to periodically update status of the *SO* within the overlay. *PONG* payload is variable according to the *PING* message received.

- *Message priority*
  For processing messages, these are processed per every *SO* according to the priority indicated in table 5.2, being 1 the topmost and 8 the lowest. This enables messages that have a more direct impact in the *SO* decision-making processes to be processed earlier than others.

5.9 Key interactions between *SOs*

This section describes how the protocol is used to carry out the key processes in which *SOs* are involved.

Table 5.2: *em4so* protocol payload descriptors

Primitive	Description	Priority
Ping	The *SO* advertises itself and check for available *SOs* to communicate with.	5
Pong	An *SO* responds to a ping message sending its own address.	4
Query	An *SO* performs a syntactical query of a role within the known *SOs*.	7
QueryHit	An *SO* playing the queried role responds to the query.	3
Execute	An *SO* asks for execution of a given action to the *SO* playing the role responsible of this action.	8
Transfer	SOs exchange information gathered by each one.	6
Committed	Having received an execute request, if the *SO* decides to execute the activity it sends back a committed message to the *SO* requester.	1
Succeed	Having executed an activity, the *SO* sends back acknowledge to the requester.	2

5.9.1 Creating/Joining the Network

The protocol enables the creation of overlay networks that become *SO* organisations including only *SOs* of interest. *SOs* that join the overlay discover a part of the participant *SOs*. A particular *SO* uses them to get access to their functionality (roles) and to other *SOs*, part of the overlay, but which are not directly connected to this *SO*. These overlays are built based on a criteria defined by the *SO* users. SOs are enabled to build or join overlays that meet that criteria. The more intuitive one is the ownership, as in the Personal Smart Spaces (PSS)[87].

In this case, *SOs* only reply back messages from *SOs* having the same owner. The joining process is depicted in figure 5.2. Every entrant *SO* must have pre configured the network address of the access member, which is currently part of the overlay it wants to join. The entrant *SO* then sends a *PING* message to the access member including as payload the join criteria to use and its introduction. The introduction is the value for the access criteria, in this case, an identifier of the owner, the pre-defined roles the *SO* is playing, the *Declared Resources Profile*, the *Current Resource Availability* and a reference to its physical location. This reference might be exact or approximated depending on how this is obtained. It could be obtained from the attached sensors —E.g. gps—, from the used protocol —if connection is made trough proximity protocols— or configured —by the *SO* Administrator—. Other criteria for access include: proximity, manufacturer, factory resources and capabilities offered, among others. These are intended to vary according to the specific domains and applications.

If the access member accepts the request, it replies back with a *PONG* message including the welcome pack and its introduction. The welcome pack is the set of updated role and scenario definitions available within the collective. The introduction is the set of roles played by the access member. The access member then forwards the *PING* message to other known members of the collective that also reply back with a *PONG* message. Periodically, the members of the overlay interchange lighter *PING* and *PONG* messages that include only the properties that have changed since the last message E.g. current available resources, referenced physical location or role offer. This interchange enables each *SO* to check for availability of others and update its *"contact list"*.

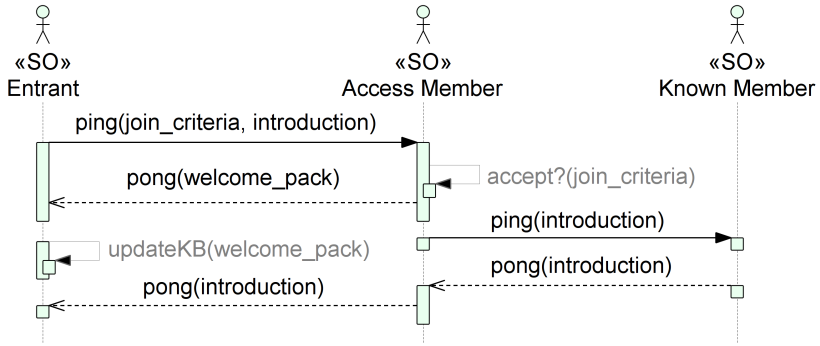


Figure 5.2: Overlay joining process

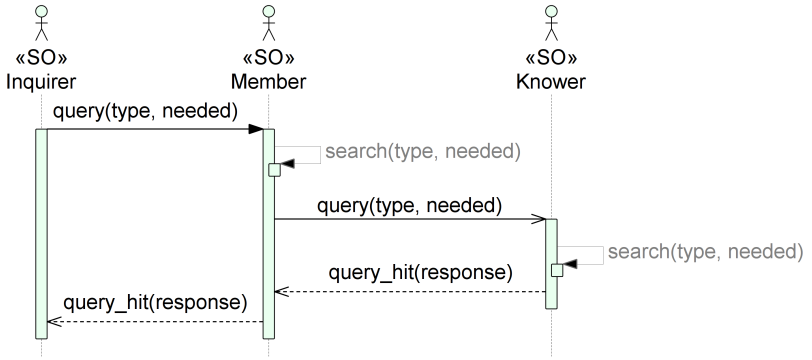
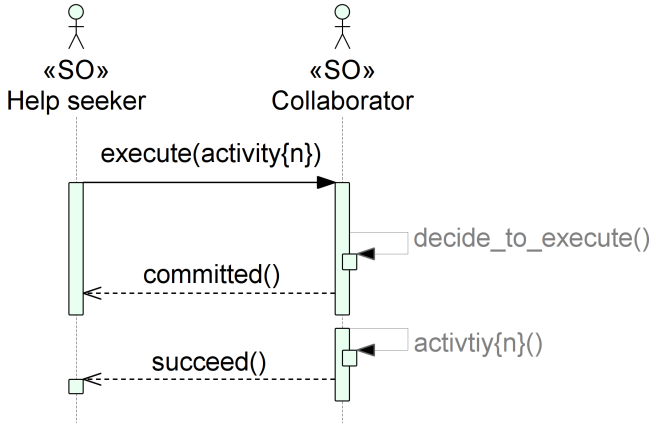
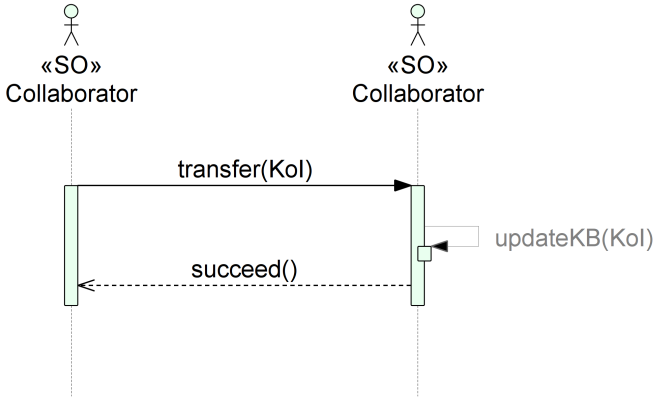


Figure 5.3: Querying within the overlay



(a) Activity Cooperation



(b) Knowledge Transfer

Figure 5.4: Supported cooperation

5.9.2 Querying within *SOs*

During decision-making *SOs* might require additional information from others to set their base of beliefs. One example is the updated status of a property of interest —from the physical environment— they are not able to sense directly. Another one, is the *SOs* able to play a role they need for a particular plan, when none of the known *SOs* is able to play it.

In these cases, the *SO* starts the process depicted in figure 5.3 by sending a *QUERY* message to the known *SOs*. Each *SO* receiving the message searches locally and reply back with the information if it has found it. For that purpose, it uses the *QUERY_HIT* message including as payload the result of the query. In the case of not finding any result, it forwards the query to others until exhausting the TTL. Every *SO* has a pre configured timeout for query operations. After timeout is reached without having received any result, the *SO* looks for an alternative course of action. E.g. wait for some configured time and then re-attempt or abort the active plan.

5.9.3 Coordination & Cooperation

The *em4so* cooperation approach is decentralised and includes both knowledge-sharing and activity-based cooperation. The activity based-cooperation is depicted in figure 5.4a. This relies in the existence of common role and scenario specifications which are ensured by the *collective organiser*. The roles, scenarios and activities are defined in each *SO*'s KB as beliefs. These are pre defined by the *SO* Administrator and updated when joining a collective.

As part of a plan being carried out by an *SO*, if it requires to ask another to carry out an activity, it sends an *EXECUTE* message with the identifier of the activity required. The *SO* receiving the request might decide to execute or not, the activity. If it decides to execute it, it sends back a *COMMITTED* message to the requester. Otherwise, it just ignores the request. After execution, the collaborator *SO* sends a *SUCCEED* message back to the requester.

During the plan execution, each *SO* has a pre defined time out in order to get *SOs* committed with activities and to get them done. If the requester does not receive the corresponding messages on time, it will attempt to carry out again, until, according to existing beliefs, it has to abort the execution of the plan and the goal. Knowledge-

sharing is depicted in figure 5.4b, an *SO* sends knowledge of interest (KoI) to other *SOs* through the *TRANSFER* message. This KoI includes beliefs that are part of one *SO*'s KB and are required by another *SO* to carry out an activity of a cooperative plan.

5.10 Implementation

A prototype of a middleware based on the *em4so* architecture was implemented. Whereas the *em4so*'s model of operation is based on agents and services, there is no constraint in the paradigm for implementing the functionality provided by the middleware and the services. The implementation requires concrete design models to be translated to the language of implementation.

The prototype design is organised around two main software components:

- **Core**
  It contains the packages and classes for the Governing and Management bodies. A structural view of the packages and their classes is shown in figure 5.5. It provides the API that is used by both the platform services and for any capability implementation.

  A closer view of a subset of the classes is presented in figure 5.6. This shows classes that give access to key services of the RE, the SOC and the KSM. SOC inherits indirectly from BaseAgent which is a subclass of the Agent class provided by the agent platform.

- **SOManager**
  This is the main thread for the runtime services. It includes classes for loading configuration files with the preferences for the *SO* and a embedded web server. The web server hosts the core component and the specific capabilities for the *SO*. This component triggers the methods of the *SO*. by referencing the core component.

The implementation was based in the Java Platform, the *EVE* Agent Platform and *CouchDB* as distributed repository. *EVE* [56], is a web-based agent platform that works on top of the Jetty Web Server. Jetty is a "small, fast, embeddable web server and servlet container" [2]. EVE promotes a model in which agents reside in a web environment, it

[2] http://www.eclipse.org/jetty/powered/

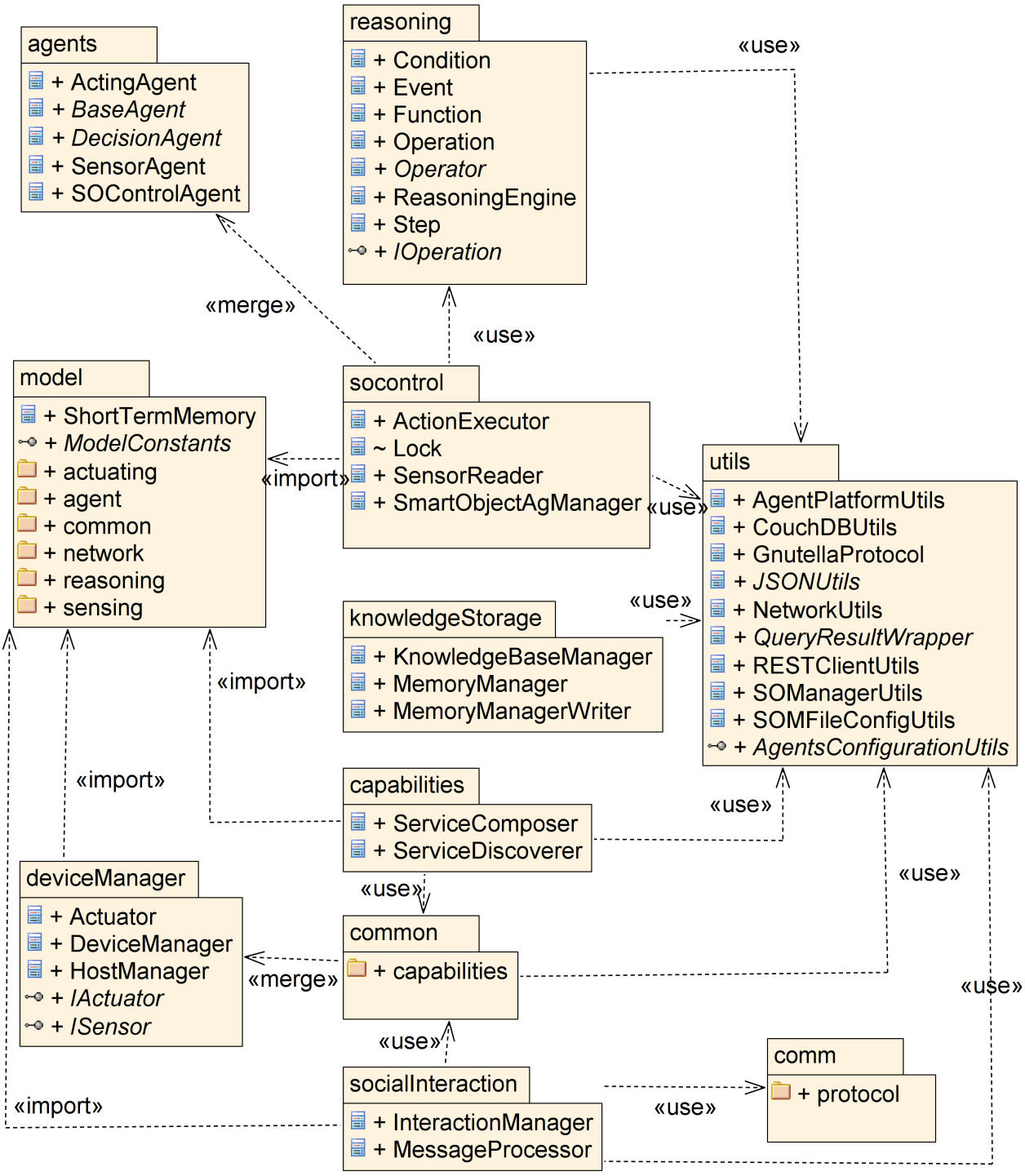


Figure 5.5: Package Diagram: *em4so* prototype's core component

does not rely in a central directory facilitator unlike most FIPA implementations. EVE is also lighter than other platforms since it offers only an essential agent model focused on communication, task-scheduling and agent memory management. These are captive features since it gives flexibility to enhance the platform using ad hoc implementations of communication protocols, agent capabilities, discovery and selection methods.

The design approach allows the *core* functionality be kept uncoupled of the web server implementation. It enables easy replacement of the middleware thread so it can be based on other server implementation. We extended the EVE platform by introducing a base agent with pre-defined features and actions common to all agents (E.g. loading the KB reference). We also included the implementation of the agent domain model with the elements mentioned in the conceptual model (Chapter 4).

For the RE, the means-ends reasoning was based on rules defined from the each *SO*'s *properties of interest* and their target values, using basic boolean algebra. The implementation of the persistence for the KB is based on *CouchDB*. It implements a document-oriented repository using JSON documents. It allows for the definition of map-reduce functions in javascript and offer a REST interface for operations over the structure and the contents. Although the EVE platform offers a simple key/value storage interface to *CouchDB*, it is restricted. So we implemented a custom client to enable more flexibility of the model for storing the status of the *SO*. The *SO* protocol is implemented on top of the EVE inter-agent communication capabilities. These services wrap agent messages as JSON-RPCs over the offered transports. We worked with HTTP since it was the most robust choice available[3].

The prototype was used for the evaluation of the architecture. Concretely, we developed the application described in the case study presented in section 7.4. Since the persistence layer is unstructured and document-oriented we defined design documents implementing the map-reduce views of the other documents in the *KB*. Hence, these views were used by the rest of components to access the *SO* data. These views include subsets of the properties of each application documents as $goals$, $observations$, $services$, $knownPlayers$, among others; and common documents representing *preconceived* beliefs. We also defined these common documents including $concepts$, $relations$, $properties$, $devices$ and $domains$, among others; that are required by *RE*

[3]Our proposal does not fundamentally depend on HTTP, so other transport protocols can be employed.

when reasoning about triggering rules. We present some examples of the map/reduce view scripts and of these documents in Appendix A.

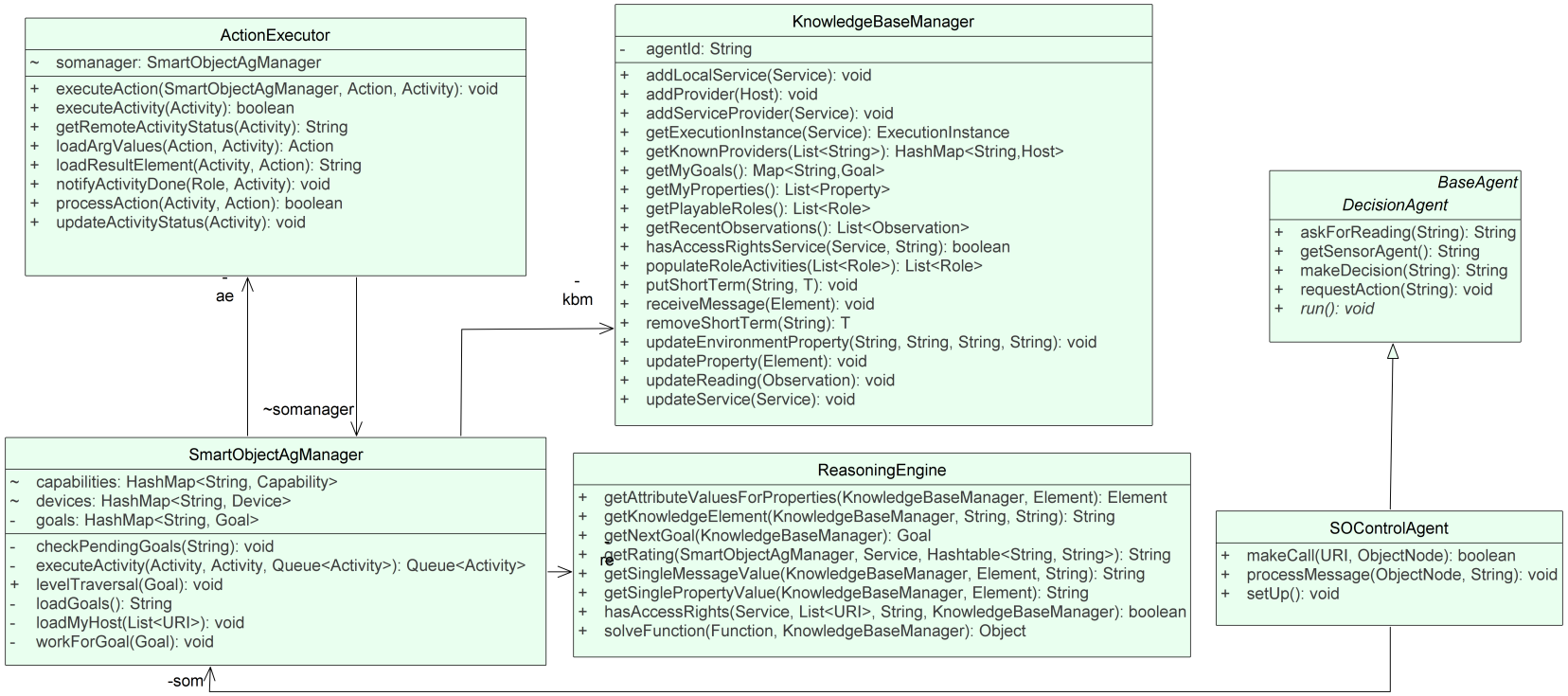


Figure 5.6: Prototype Middleware: Class Diagram Excerpt

5.11 Summary

This chapter has introduced the middleware architecture we propose covering the common functions of the software engineering approach for autonomous and adaptable *SOs*. The architecture is inspired in an enterprise organisation and components are arranged around two bodies —Governing and Management— and a set of Communication, Device and Extra facilities.

Two of the key components are the Capability Manager and the Social Interaction Manager. The first one has two purposes: as a provider, loading and unloading of the capabilities at runtime, as a consumer, it localises and binds the services required in order to carry out the *SO*'s goals. The second one, manage the communication and carries out individual and collective adaptation based on the established social relations.

We have described the *SO* protocol used to carry out the main interactions among *SOs* including joining a collective of *SOs*, querying services offered and seeking for cooperation. The protocols is based in the Gnutella P2P and enable the creation of unstructured overlay network for the collective of *SOs*.

We have implemented a prototype using the proposed *em4so* architecture. The prototype demonstrated the feasibility of the architecture using available platforms and tools.

Chapter 6

Adaptation of *SO*-based IoT Systems

6.1 Introduction

The vision of *SOs* and the *SO*-based systems includes the existence of self management abilities, i.e. act as an autonomic system. Self management capabilities implies the autonomy by the *SO* systems to make decisions in regards how they operate and approach their goals. We have seen that one of the key process for a relative autonomy is the generation that might be achieved by means of adaptation. Adaptation as defined in Cambridge dictionaries is "the process of changing to suit different conditions"[1]. Since adaptation is a common function that is required for multiple *IoT* applications and is instrumental for achieving autonomy, it makes sense that *em4so* middleware architecture provides an approach for adaptation that takes advantage of its functional components.

In this chapter, we present the adaptation method used by the *em4so* middleware architecture. In section 6.4, we first present the strategy we use in order to provide adaptation functions to the *SOs* built on top of the *em4so* middleware architecture. We also distinguish the types of supported adaptation and explain the drivers and how the selection of the *SOs*, services and roles are the basis of the proposed adaptation mechanism. Next, we apply the presented strategy to the selection of *SOs* (section 6.5), services (section 6.6) and roles (section 6.7). We present the details and particularities of each selection process and present the functions and algorithms used.

[1] http://dictionary.cambridge.org/dictionary/english/adaptation

6.2 Research Challenges and Requirements

The *SO* systems might adapt to multiple situations at both individual and system level. These situations involve factors regarding the *SOs*, their resources, the cyber physical and social environment and the human users, among others. Since it is not realistic to cover all the possible situations that would require adaptation, the main requirement is that an *SO* middleware solution provides a general approach for adaptation that can be extended to each particular application. This approach should cover some of the typical situations that we summarise in the following requirements:

- The *IoT* system must be able to operate with different configurations of *SOs*. The approach for achievement of system goals, either individual or collective, must work regardless of the specific available *SOs*. Equivalent *SOs* might have different characteristics but must share the ones that make them suitable for co-operating with others in achieving a common goal.
- The *IoT* system must be aware of and cope with the unavailability of particular *SOs* avoiding or reducing the disruption in the achievement of the system goals. Typical volatility of the *IoT* environments must be managed ensuring that a system of *SOs*, as whole, is able to recover and overcome failure, departure of individual components, namely, *SOs*.
- The *IoT* system must deal with individual *SO*'s variability. *SOs* might change their characteristics while available because of several reasons, for example, mobility, resource exhaustion and behavioural or structural changes. These changes might have impact in the suitability of these *SOs* for achieving the system goals. The system must detect these changes and ensure a proper allocation of task within the available *SOs*.
- *SOs* must identify opportunities to save local resources and adapt its behaviour and structure accordingly.

6.3 Contributions

The main contribution of this chapter is to provide a decentralised method for the adaptation of *SO*-based *IoT* systems. This method takes advantage of the *RbSOs* and *em4so* architectures and also uses multi-objective optimisation for selecting *IoT* services. Our contribution is in using these known techniques for a decentralised adaptation (See section 2.3.5), based on the allocation of *SOs*, enabling *SO* collectives to adapt to variable

type and quantity of *SOs*, to the lack of availability of some of them and to changes in the internal *SO* structure and behaviour. Service selection and multi-objective optimisation have been used previously in *IoT* context for allocation of services and nodes. However, there is a lack of dynamic decentralised solutions that work under the concept of smart object as autonomous entities (See section 2.5). In this sense, we incorporate the use of roles and grouping of selection factors enabling us to propose:

- A strategy for adaptation of *SO* systems based on the dynamic and decentralised selection of *SOs*, services and roles and that considers changes in some of the key factors of the *SO* systems and its context.
- Definition of utility-based methods for dynamic selection of *SOs* based on roles and a localised/decentralised decision-making process.
- Definition of utility-based methods for adaptation of the *SO*'s internal structure based on services and roles and a localised/decentralised decision-making process.

6.4 *em4so* Adaptation Strategy

Runtime adaptation implies the ability of the *SOs* to make decisions according to the current circumstances and based in a defined criteria. The main operation of the *SOs* comes from the execution of the plans for the achievement of a defined goal. These plans involve sequential or parallel activities to be carried out by one or multiple *SOs*. In the case of goals requiring cooperation of multiple *SOs*, plans provide the interaction workflow between the different role-playing *SOs*. From the triggering of an activity to the moment the individual actions, part of the activity, take place there are different decisions taken by participating *SOs* that make execution flexible and adapted to current situation.

Adaption involves decision-making which is, ultimately, the selection between different alternatives. If these alternatives are assessed based on parameters that describe them and the relevant context properties, the alternatives become sensitive to changes in the context. Therefore, if selection is based on this assessment the best suited alternative will change according to the context. For adaptation of the *SO* systems, these alternatives present the possible configurations, the task is to select the best fit for the context and the user requirements. *em4so* middleware enables adaptation at collective and individual level.

At collective level, a configuration is the set of different *SOs* that carry out a plan from the available in the collective. At individual level, a configuration is the different mix of services and roles that are chosen by a particular *SO* according to the situation. Since an *SO* collective is not centrally controlled by a particular *SO* or a platform, the decisions that shape the collective configuration are made individually by each *SO*. *em4so* ensures that a common model for adaptation decision-making is used among the different cooperating *SOs*.

There is a trade-off between decentralisation of decisions and the success of collective plans. On one side, the decentralisation brings independence in the adaptation and resilience to the *SO* systems, however it makes an *SO* collective unable to always ensure the achievement of the common goals. This is mainly due to the partial knowledge that every decision-maker (*SO* member) has, particularly, when finding another *SO* to cooperate with. Although the *RbSOs* architecture enables prioritisation of collective over individual goals (See section 4.4.2), the lack of knowledge might lead to decisions that are not optimal for the collective.

One example is when an *SO* aims to save energy, this is an individual goal that, at first, seems also beneficial for the collective as it helps to maximise availability of their *SO* members. However, this goal can lead to reduce the services offered by an *SO* within the collective, therefore other *SOs* are unable to request execution of a needed activity. Although our approach incorporates interchange of information, by *SOs*, about their available resources, finding the optimal solution for the collective can not be guaranteed. In this case, it depends on the *SO* collective administrator to determine alternative *scenarios* when common goals can not be achieved. To mitigate this situation, the *SO* manages its *"contact list"*, by favouring the *SOs* it needs (to achieve a common goal) and that have had successful interactions with it (See section 5.6.2). Considering a long-term view of dynamic *IoT* scenarios and where there are multiple *SOs* offering equivalent services, we consider our strategy is acceptable.

6.4.1 Collective Adaptation

The collective of *SOs* is an *IoT* system composed by the set of nodes —the *SOs*— that are connected through an overlay network enabling them to interact and cooperate between each other. The system achieves its goals by means of the *SOs* that are part of

it. These *SOs* carry out the actions that collectively lead to the goal achievement. The *em4so SO* protocol (Section 5.8) provides the mechanism for sharing and gathering data between the *SOs*, part of the collective.

For a plan execution, there is not a unique *SO* being able to carry out an activity, on the contrary the *SOs* offer redundant services that end up completing the required activity. In addition, the *SOs* are not always connected and they might move from one space to other making their services unavailable or less suitable for a task. Therefore, *IoT* systems can not rely in specific *SOs* to achieve their goals. Considering these characteristics, our approach is to enable each *SO* to, autonomously, select among other various *SOs* to cooperate with, according to the current conditions. Since these conditions are changing, every time each *SO* needs to find an *SO* to cooperate with, it can check if the conditions have changed and so find a better suited *SO* for the new conditions or if the conditions remain, but the *SO* is no longer available, identify the best substitute.

In *em4so* middleware architecture, the uncoupled goal-motivated behaviour of the *SOs* (Section 4.4.2) together with the *SO* protocol (Section 5.8) endow flexibility to the *IoT* system to carry out plans without depending in concrete *SOs* and following the distributed peer-to-peer approach without requiring a central coordinator. Hence, roles enable decoupling of the responsibilities and the functionalities from the entities playing the role. In addition, in *em4so* roles are also decoupled from the specific workflows leading to the achievement of goals, which are defined by the plans. These characteristics set the basis for the collective adaptation as plans are defined functionally in terms of roles whose players are chosen dynamically according to the execution context. Over this basis, the collective of *SOs* faces one fundamental adaptation question: which *SO* should carry out the next activity of a plan given the current context?

6.4.2 Individual *SO* Adaptation

The *SO* from an individual point of view is an entity that has been delegated some work to do (expressed in plans), as per the roles and capabilities it has. The *em4so* middleware architecture is designed to enable the *SO* to carry out the work in different ways according to the user preferences and context. The selection of capabilities (services) to be used for this work is dynamic, as these are evaluated when the activities are triggered in order to detect changes that might require a different configuration of services for an

activity. Besides deciding in which particular goals to engage, *SOs* make decisions for adapting elements in its key areas —E.g. structure and behaviour— based on these preferences and situation. Thus, *SOs* face three key adaptation questions they need to make a decision about:

- Which services should it use to carry out an action within an activity and plan?
- Which services should it deploy/undeploy in a particular moment?
- Which roles should it play in a particular moment?

The decisions to answer these questions are made at different moments of the *SO* life cycle. Whereas the first one is made when the *SO* has committed to an activity, the others are made in a regularly basis as part of the *SO*'s maintenance goals. In addition, since the autonomy is relative, decisions made might have impact in the autonomy of *SO* for a particular goal and situation.

6.4.3 Adaptation Drivers

The adaptation as whole, including the selection of the *SOs*, services and roles according to user preferences and context, is motivated by a set of decision drivers. The first decision driver is the functionality, i.e. the *SO*, to pick for an activity, must have configured the corresponding role that includes the required activity. Besides the functionality, non-functional properties i.e. *QoS* attributes, are used to select among available candidates —e.g. various *SOs* playing the same role or various service providers—. We categorise the key non-functional factors to consider in the context of *IoT* applications in groups. The purpose is to ease the configuration of preferences for these factors and enable comparison of the relative importance of each group to the user.

For describing our adaptation approach we identified four reference groups where the most relevant attributes for *IoT* applications can be gathered: trustworthiness, performance, efficiency and context. Additional groups can be added without affecting significantly the approach, however, the more groups identified, the less the advantage of grouping factors. We assign every non-functional attribute to one group. For example, the context group is aimed to gather the most diverse requirements that are specific of the context where the *SO* is being used, e.g. location and usability. On the other side, the trustworthiness group includes, for example, security, privacy and safety factors. These groups are presented in figure 6.1 and cover characteristics of the *IoT* applications that might vary and to which the *SOs* have to adapt.

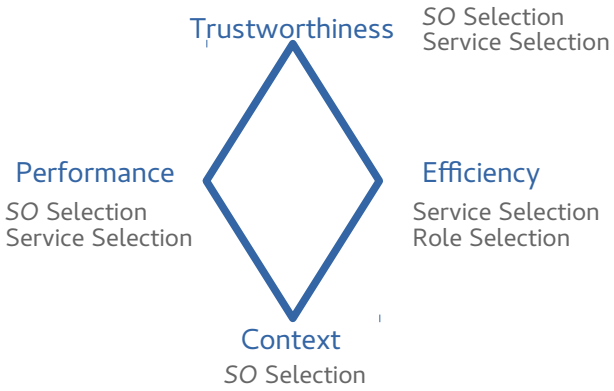


Figure 6.1: Group of Factors Involved in *SO*'s Decision-Making

The different selection processes address some of these group of factors and together these cover all of them. The group of factors are the basis for the definition of selection criteria for each selection, thus enabling the dynamic comparison of the different alternatives and finally leading the adaptation process. Each group includes a number of metrics that can be calculated in order to compare different alternatives. We model each selection process as separated optimisation problems whose objectives are defined from the group of factors identified. The selection of the *SOs* is a multi objective problem where objectives are defined by *trustworthiness*, *performance*, and *context* factors. The selection of services is also multi objective with objectives given by *trustworthiness*, *performance*, and *efficiency* factors. Finally, the role selection is addressing the single *efficiency* group. Each group of factors is defined by an utility function that includes the assessment of each alternative using the metrics defined for the group. According to each case, the aim is to maximise or minimise, the utility function in order to get the best alternative.

6.4.4 Multi Objective Optimisation

For multi objective problems we use the *Simple Additive Weighting (SAW)* method [26]. It proposes the definition of an aggregated utility function from the weighted sum of the assessments of the alternatives according to each individual objective involved. The best candidate comes from maximising the aggregated utility.

In our case, each individual objective corresponds to the group utility of each candidate —i.e. *SO* or service, according to the case— for the group of factors. Thus we can define the group utility function $ug_{i,j}$ of a candidate c_j for each group g_i using the

arithmetic mean of the factors of the group:

$$ug_{i,j}(c_j) = \bar{a_{l,j}}(c_j) = \frac{1}{n}\sum_{l=1}^{n} a_{l,j}(c_j), \tag{6.1}$$

where $a_{l,j}(c_j)$ is the function that returns the normalised value of the metric l for the candidate c_j and n is the number of metrics of a group g_i. Since we want to avoid having to define a weight for each factor of the group, we leave this as a simple mean, however, it can also be replaced with a weighted mean then, requiring the user to specify the weight for every factor.

To resolve situations where two or more alternatives have identical values of utility, we incorporate the use of rankings for determining the weights. These rankings, define the order of relative importance, to the user, of each key factor involved in each selection. This simple approach has a number of advantages:

- Each group utility is a meaningful measurement of the related metrics, enabling to assess individually each optimisation objective.
- It avoids the need of defining preference weights at lower level, E.g. device characteristics or *QoS* attributes. This is consistent with the autonomous *SOs* as the user should be able to define preferences at a higher level without the need of atomic decisions that are left to the *SO*.
- The consolidated utility shows the dominance of the groups of factors avoiding conflicts for *nondominance* of criteria.

For a multi objective problem, we define g as the set of groups of factors such as $g = \{g_1, g_2, ..., g_k\}$, where k is the number of groups identified. For each group, the user defines the vector r, which includes the set of the rankings of the relative importance of the ith group g compared to the other groups: $r = \{r_1, r_2, ..., r_k\}$. Therefore, we can define the user preferences in regards the relevant groups of factors for a selection problem as the $2 \times k$ matrix m:

$$m = \begin{bmatrix} g_i & g_{i+1} & ... & g_k \\ r_i & r_{i+1} & ... & r_k \end{bmatrix} \tag{6.2}$$

In practice, the *SO* Administrator specifies the mentioned preferences per user.

In order to apply the *SAW* method, we need to transform the rankings r in normalised

weights. To do so, we use:

$$\omega_i(r_i) = \sum_{i=1}^{k} \frac{1 + k - r_i}{\sum_{d=1}^{k} d}, \tag{6.3}$$

where ω_i is the equivalent normalised weights for the rank r_i.

At this point, we can use the normalised weighted mean to obtain the aggregated utility u_j for each candidate c_j:

$$u_j(c_j) = \sum_{i=1}^{k} w_i \cdot ug_{j,i}(c_j) \tag{6.4}$$

Thus, we pick the candidates with the maximum utility. In case of candidates with the same utility, we use the group utility as criteria for settle the final decision. In the following sections we present how this approach is used for selection of *SOs* and services.

6.5 Selection of *SO*

This process starts with a *source SO* triggering a plan. It carries out the activities it is responsible for and then finds and selects a *target SO*, that is able to carry on executing the plan, until the plan is complete and the goal is achieved. In this process, the *source SO* might have in its *"contact list"* an *SO* playing the required role or, since this list is reduced and there is no room to every *SO* in the collective, it might need to query for it (Section 5.9.2).

Once some candidates are found, the *SO* compares them to determine which one is the best *SO* to control the collaborative plan, according to the preferences and context. Then the *source SO* sends a request for activity execution to the *target SO*. As the *SOs* are autonomous, the *target SO* might decide to commit to an activity or just ignore the request. Ignoring the request causes that the *source SO* look for another *SO* to pass the next activity on. Once an *SO* has committed to carry out an activity this process is restarted from the *target SO* until the goal is achieved.

The aggregated utility for *SO* selection covers the groups of *trustworthiness*, *performance* and *context* (figure 6.1). We have chosen the functions of *reliability*, *potential performance* and *estimated distance to target* to represent each group respectively.

These functions are based on the metrics gathered by *em4so* middleware for each *SO*. Therefore we can particularise Equation No. 6.4, with:

$$u_j(so_j) = w_t \cdot rel_j(so_j) + w_p \cdot pp_j(so_j) + w_l \cdot edt_j(so_j) \tag{6.5}$$

where the weights come from the rankings specified by *SO* Administrator for each group. For example in a matrix such as:

$$m_{so} = \begin{bmatrix} t & p & l \\ 1 & 3 & 2 \end{bmatrix} \tag{6.6}$$

The *SO*'s *Reliability* is calculated from the previous interactions the source *SO* has had with potential *SO* candidates. Every time the *SO* interacts with another *SO*, *SIM* middleware component (Section 5.6.2) keeps track of these, for example recording requests, commitments and success of the other *SOs* in regards a plan activity. The *SO*'s *Reliability* can be calculated with:

$$rel_i = \alpha \frac{as_{j,i}}{ac_{j,i}} + (1 - \alpha) \frac{ac_{i,j}}{ar_{i,j}}, \tag{6.7}$$

where $as_{j,i}$, $ac_{j,i}$ and $ar_{j,i}$ represent the previous successful, committed and requested activity executions, respectively, from an SO_j to an SO_i. α is a constant that enables to vary the importance given to the successful execution after commitment or to the commitment itself after requesting. If there has not been any previous data yet, the SO_j uses the data provided by another trusted *SO*, or in absence, it uses a default arbitrary score, for example 0.5.

The *Potential Performance* (*pp*) gives an indication of the amount of resources available in the candidate *SO* and works only for determining which *SOs* have better hardware configuration than others. The assumption is that the better the *SO*'s hardware configuration the better the performance in the execution of a plan activity. Since the *SO*'s has a mix of *Transient* and *Persistent* resources (Section 3.6.4), the *pp* is calculated differently according to the resource usage patterns. For *Persistent* resources, as these are stable, we use both the *Declared Resource Profile* (*DRP*) and the last *Current Resource Availability* (*CRA*), reported by the *SO* through the *SO* protocol (Section 5.9.1). For *Transient* resources, as these change very quickly, we use only *DRP*. In practice, *DRP* is a percentage (set by *SO* Administrator) of the real *SO*'s hardware resources.

We emphasise the *Potential* nature of this component as it does not consider the current *SO*'s workload that also have impact in the performance. Although having better hardware configuration —E.g. more CPU power or better sensor quality— generally derives in a best performance it also attracts more work, as other *SOs* might choose the most powerful *SOs*, implying then a higher workload for an *SO* compared to others with low potential resources. However *DRP* is a stable measurement which is more reliable than an instant view of the available resources, that for sure, will have changed when an activity is finally triggered. The *pp* is not used for a precise work allocation as it is not an exact, but an estimated, measurement based on the data shared by the candidate *SO* about its hardware platform. Having a high workload might be one of the reasons, of a *target SO*, for ignoring an activity request. The *pp* for an *SO i* is calculated from:

$$pp_i = \frac{1}{2}\left(\sum_{j=1}^{kt} tre_j(drp_{i,j}) + \sum_{l=1}^{kp} pre_l(drp_{i,l}, cra_{i,l})\right), \tag{6.8}$$

where $tre_{i,j}(drp_{i,j})$ and $pre_{i,l}(drp_{i,l}, cra_{i,l})$ are the functions that calculate the normalised value, considering all the *SO* candidates, for each *Transient* resource j and *Persistent* resource l, respectively, from a total of kt *Transient* and kp *Persistent* resources. The arguments $drp_{i,j}$ is the corresponding declared resource profile, from *DRP*, for the resource j and $cra_{i,l}$ is the last known available measurement, from *CRA*, for the resource l. The particular functions to calculate $tre_{i,j}$ and $pre_{i,l}$ are not fixed and depend on each particular resource. These might be as simple as returning the normalised value or include more calculations.

Consider a simple example with memory (m), CPU (c) and battery (b) as resources and the following functions:

$$tre_{i,m}(mem) = \frac{mem_i}{\sum_{s=1}^{n} mem_s}, \tag{6.9}$$

$$tre_{i,c}(cores, cspd, cch) = \frac{1}{3}\left(\frac{cores_i}{\sum_{s=1}^{n} cores_s} + \frac{cspd_i}{\sum_{s=1}^{n} cspd_s} + \frac{cch_i}{\sum_{s=1}^{n} cch_s}\right), \tag{6.10}$$

$$pre_{i,b}(bac, bav) = \frac{bac_i \cdot bav_i}{\sum_{s=1}^{n} bac_s \cdot bav_s}, \tag{6.11}$$

where mem, $cores$, $cspd$, cch, bac and bav are vectors of $1 \times n$ representing the memory size, number of cores, the CPU speed, the CPU cache size, the battery capacity and

the available battery of the *n SO* candidates; and the sub index *i* represents the value for the ith candidate.

For describing the *context* group we are going to consider only the physical location, however other attributes can be included. The *Estimated Distance to Target* (edt) is a normalised value, calculated from the physical location information that the *SOs* share when joining the overlay network. This is updated regularly via the *SO* protocol, for the *SOs* that are part of the "contact list" of an *SO*. Part of the properties of the *SO* include its physical location. There are *SOs* that have the capabilities backed by sensors —E.g. GPS— enabling them to report its own location, whereas there are other *SOs* that are not able to. In the latter case, we assume the location of the *SO* is not relevant for the *IoT* application or the *SO* is static. Hence, in this situation, the *em4so* middleware requires the *SO* Administrator to specify the location via a parameter.

By default the *SOs* will try to locate other *SOs* that are closer to a given target E.g. the user, the *source SO* or a reference point. This reference point is by default the user, but can be overwritten as the input knowledge of each activity. So this metric indicates how close the *SO* is from the target. This calculation depends on the precision of the location reference available, E.g. outdoor absolute coordinates as GPS, or indoor relative location. There are multiple algorithms and methods for calculating position and distance, authors of [4], present an example of calculating distance from GPS readings. These methods can be implemented as *Extra Facilities* (Section 5.7.3) or just consume from existing web services that have these methods already implemented[2] and convert to required units. Therefore we assume we obtain the distance to the target of each *SO* candidate and normalise it with a simple quotient:

$$edt_j(so_j) = \frac{dis_j}{\sum_s^n dis_s}, \tag{6.12}$$

where dis_j is the distance to the target of and SO_j over the sum of the distances of all the *n SOs* candidates. The Algorithm 2 shows the process for the selection of an *SO*.

6.6 Selection of Services

An *SO*, as an autonomous entity, controls the invocation of the services it uses to carry out the job required for an activity and plan. The activity is an abstract service def-

[2]https://developers.google.com/maps/documentation/distance-matrix/start

inition that, in order to be executed, needs the services to invoke with the concrete requirements for the actions adapted to user preferences and current context. Every *SO* has a number of capabilities —realised as stateless services— that are locally —on-object— available so the *em4so* middleware is able to manage them. These capabilities are the main source for actions carrying-out an activity to which the *SO* has acquired a commitment. However, as usually the *SO* is connected to the cloud it might be the case that other remote service providers are available for the actions. Hence, the *SO* must decide which service to consume for the action, considering the available offer.

```
input: A matrix of user's preferences: mPreferences, Set of pending
       activities to trigger for this SO: pendingActivities, A map
       indicating the queries that have been sent by activity: sentQueries
1  /* The "contact list" is updated asynchronously
      when other SOs respond to query messages.   */
2  while pendingActivities ≠ ∅ do
3      activity ← getFirst (pendingActivities) ;
4      candidates ← ∅;
5      theRole ← getResponsibleRole (activity);
6      foundCsos ← queryLocal (theRole) ;   // Check "contact
        list"
7      if foundCsos ≠ ∅ then
8          for cso in foundCsos do
9              gUtilities ← calculateGroupUtilities(cso) ;
                // (6.1)
10             utility ← calculateAggrtdUtility(gUtilities) ;
                // (6.4)
11             candidates ← add(candidates, cso, gUtilities, utility) ;
12         end
13         selectedso ← maxUtility(candidates, mPreferences) ;
            // 1) by aggregated utility 2) by dominance
            order
14         sendExecuteMsg (selectedso, activity);
15         pendingActivities ← remove (activity);
16     else
17         if contains (sentQueries, activity) ≠ true then
18             sendQueryRoleMsg (theRole) ;      // Send query
                message to known SOs (Section 5.9.2)
19             sentQueries ← addQuery (sentQueries, activity)
20         end
21     end
22 end
23
```

Algorithm 2: *SO* Selection

The flow for allocation of services starts when the *SO* has committed to do an activity. The services are bounded the first time these are used, so the subsequent times, the bounded services are used instead of doing a full search and selection process. The *SO* gathers, through *CM* middleware component (Section 5.6.1), the current state of the preferences and the context data required for the activity. The bound services are only updated if there are changes in the requirements or the availability of the service.

The selection problem is slightly different to the *SO* selection presented before. One particularity is that the number of services that might be potential candidates is higher than the number of *SOs* that are part of a collective. The service selection has been tackled from different perspectives and researchers have overcome this problem with different strategies. Our search and selection strategy is based in the pre-selection of a reduced set of services in order to then selecting over it the best fit for the service. This strategy has been previously applied, by service selection researchers, for example in [5, 67], with different calculation methods and criteria for making the pre-selection. We subsume this approach as an extension of our base adaptation strategy presented in section 6.4.3.

Another particularity of the service selection is that usually preferences, as well as concrete non-functional requirements, for each service are specified as values of *QoS* attributes. Service selection researchers have identified and proposed a vast number of *QoS* attributes and methods for calculation and assessments. Although having a fine coarse grained definition of preferences and multiple *QoS* attributes with sophisticated calculation methods might fit concrete needs and give more control to the user, in practice, it evidences a lack of delegation towards the *SO*. In addition, the more complex the calculation method the more time it takes to gather arguments for calculation and selection which is challenging in constrained resources objects.

We organise the *QoS* attributes in the groups of the factors previously presented (figure 6.1). Some examples of concrete *QoS* attributes are:

- *Trustworthiness*: privacy level profile, reputation, reliability, security profile, etc.
- *Performance*: response time, precision, range, accuracy, etc.
- *Efficiency*: CPU usage profile, storage usage profile, energy usage profile, etc.

Note that many of the performance attributes are specific for the type of service, for example, services backed by a physical sensor might be described with attributes about

the precision of the sensor but also the reading range, which are assumed to be arguments of the service. Note also this list is not exhaustive, as we said, multiple other attributes exist an so the methods for calculating them. The more the *QoS* required to consider, the more values will need to be specified by the user for its preferences. Each *QoS* attribute is classified by the CM according to the *preconceived beliefs*.

For service selection, besides the relative importance of each group of factors we define the tolerance level for a set of related *QoS*. This avoids the need of defining tolerance for each particular *QoS* attribute. Thus, we extend the matrix m presented in Equation No. 6.2 with another row h representing the normalised tolerance for the specified value in each *QoS* value of a group g: $h = \{h_1, h_2, ..., h_k\}$. As a result, we can define the general preferences of an *SO*'s user as the $3 \times k$ matrix m':

$$m' = \begin{bmatrix} g_i & g_{i+1} & ... & g_k \\ r_i & r_{i+1} & ... & r_k \\ h_i & h_{i+1} & ... & h_k \end{bmatrix} \tag{6.13}$$

Thus, for *SO*'s service selection the three important groups of *QoS* attributes are: $g = \{t, p, e\}$ with t,p,e *trustworthiness*, *performance* and *efficiency* respectively. Hence, one example of the concrete preferences of an user can be defined with the 3×3 matrix m'_{so}:

$$m'_{so} = \begin{bmatrix} t & p & e \\ 2 & 3 & 1 \\ 0.2 & 0.5 & 0 \end{bmatrix}, \tag{6.14}$$

where the most important factors are *efficiency*, *trustworthiness* and *performance*, in that order, with *efficiency* as the *dominant factor*. Likewise, it indicates that the user has no tolerance for the *QoS* attributes of *efficiency* but is relaxed about *performance* attributes.

Calculation for *trustworthiness* and *efficiency QoS* attributes is common for every service, so we can define it. The values for the *trustworthiness* group are initially calcu-

lated with a simple ranking as follows:

$$a_t(s_j) = \begin{cases} 1 & s_j \text{ is locally hosted} \\ 2 & s_j \text{ has } \textit{High reputation} \\ 3 & s_j \text{ has } \textit{Medium reputation} \\ 4 & s_j \text{ has } \textit{Low/Unknown reputation} \end{cases} \tag{6.15}$$

In the case of remote services, we assume the Service Directory records reputation based on the scores given by other service users after interaction. After the first use, the *SO* records the state of services attempted and their response, thus enabling calculation of *reliability* in further uses.

The *QoS* attributes for *efficiency* are defined mainly in terms of usage patterns for resources. For locally hosted services *SOs* record the usage of the key resources (CPU, battery and storage) every time a service is invoked, this information is used in further selections. We are describing decisions made from the *SO* perspective, it means that *efficiency* only takes into account the resource usage of the individual *SO* instead of a whole collective and remote services.

Hence, the resource usage of remote services only considers the resources for invoking the service, sending the required data arguments through and receiving the response. This enables, that by means of *Extra Facilities* the *SO* is able to offline classify — E.g. using the *k-means* algorithm [53]— every observation of resource usage for all the services, to fit a reduced group of *usage patterns* (E.g. *high*, *medium* and *low*). This classification provides a relative scale to compare services, according to the resource usage. Thus, *SO* is able to autonomously gather the service usage patterns and tune them to the underlying hardware platform efficiency. For example, if the energy consumption (battery degradation for a battery-powered *SO*) of using the network interface for requesting/getting response to/from a remote service is low, it makes the *SO* select these services over the local hosted. On the other hand, if the energy consumption for the same task is the same or higher, compared to running a service locally, the decision is clear on invoking the local service, if available. Therefore, *efficiency* based on the

usage patterns can also be simplified to the ranking:

$$a_e(s_j) = \begin{cases} 1 & s_j \text{ has } \textit{Low} \text{ resource usage} \\ 2 & s_j \text{ has } \textit{Medium} \text{ resource usage} \\ 3 & s_j \text{ has } \textit{High} \text{ resource usage} \end{cases} \tag{6.16}$$

For the *performance* group, another ranking might have been specified, however it has less value as the *QoS* attributes for assessing each service's performance are highly heterogeneous, so it is more suitable to define performance in regard to every particular service domain. As the set of candidate services might be large, we firstly pre-select a reduced set, working only with the group g with the highest ranking. This group is generally the most restrictive (less tolerance) group also, ensuring a well reduced set as the starting point. With the reduced set we calculate the remaining group utilities. The algorithm No. 3 shows the process.

6.6.1 Selection of Deployed Services

There are contexts where local capabilities might not be required due to availability or more suitable remote services. For example, when sensor precision is the key *performance QoS* requirement and the sensors available for the *SO* do not meet the expected level. Once capabilities are deployed, these consume resources that might be allocated to other activities. Since these capabilities are realised as service components, these can be deployed and *undeployed* in runtime. Thus, it is advantageous to undeploy local capabilities that are not used, so the *SO* adapts its structure to the existing service offer.

The CM is in charge of periodically check for capabilities that are candidates to be *undeployed*. The selection process presented in Algorithm 3 identifies every time the local capabilities are discarded to be used in an action (line 17). Besides just marking the capability, this method stores the current state of the *SO* when the capability is discarded (physical location, connected network, time and active user). The default operation of the CM *undeploys* the capabilities that have been marked unused a number of times since the last deployment. Both the frequency for these checks and the number of times required to disable the capabilities are parameters defined by *SO* Administrator. Further classification of the data recorded is intended to be performed using *Extra Facilities*. The implementation of classification algorithms to identify the *discarding patterns* is subject to the particular *SO* solution and the underlying hardware platform.

```
input: A matrix of user's preferences: mPreferences, the set of actions
1  for action in actions do
2      boundService ← getBoundService(action);
3      if boundService == ∅ or
          checkContextRulesChanges(action) == true or
          ping(boundService) == false then
4          candidates ← ∅;
5          serviceContract ← getServiceContract(action, mPreferences);
6          localService ← getLocalService(capabilities, serviceContract);
7          if localService == ∅ or
              isLookForRemote(localService, mPreferences) == true
           then
8              dGroup ← maxRankingVector(mPreferences);
9              dominantQoS ← getQoSGroup(action, dGroup);
10             preSelected ← lookUpService(serviceContract, dominantQoS);
11             for pres in preSelected do
12                 gUtilities ← calculateGroupUtilities(pres) ;
                    // (6.1)
13                 utility ← calculateAggrtdUtility(gUtilities) ;
                    // (6.4)
14                 candidates ← add(candidates, pres, gUtilities, utility);
15             end
16             boundService ← maxUtility(candidates, mPreferences) ;
17             if boundService ≠ localService then markNotUsed(capabilities,
                localService);
18         else
19             boundService ← localService;
20         end
21     end
22     invokeService(boundService, getArguments(action),
        returnToElement(action));
23 end
```

Algorithm 3: *SO* Service Selection & Invocation.

6.7 Selection of Offered Roles

Every *SO* has a set of roles to play that is predefined (before runtime) by the *SO* Administrator. These roles represent the commitments an *SO* is willing to take and also the constraints on its behaviour, imposed by the *SO* Administrator. The roles arrange a set of activities (composite services) and actions (atomic services) that are generally realised by local capabilities and, in some cases, by remote services. Although it is intended that roles are stable and that every *SO* plays a reduced number of roles, from an autonomic perspective, roles fulfil the purpose of enabling the management of *SO*'s behaviour and structure at a higher level, in order to adapt to the context while pursuing *SO*'s optimisation goals. The management at role level has two main advantages: *a)* it provides a functional association among multiple services; and *b)* Since roles are intended to be fewer than the capabilities available, the effort of managing a more reduced set of *autonomic artefacts* is lower than that of managing individual capabilities.

The following example helps to illustrate the concepts around the *SO* role management. A decorative and battery-powered *SO* "smart flowerpot" (*SFP*) monitors the plants it contains and uses its led panel and incorporated speaker to notify to the user about the plants status and its estimated needs. Depending on the hardware platform *SFP* might play multiple roles, minimally *"Soil Carer"* and *"User Mediator"* and optionally others linked to its physical capabilities such as *"Music Player"* and *"Fire/Smoke Detector"* or others more general such as *"Music Recommender"*, *"News Feed aggregator"* and *"Unit Converter"*. The role *"User Mediator"* includes activities for formatting and transforming raw data in a layout able to be presented, according to the particular user interfaces available (E.g. wide/tiny screens, speakers, leds, etc). *SFP* has its own implementation of activities of the *"User Mediator"* role using its led panel and speaker to present alerts and interact with end users. The role *"Soil Carer"* includes activities to monitor the physical properties of the soil and recommend or take actions to improve its physical conditions. *SFP*'s implementation of this role uses its local sensors (temperature, humidity, light, movement and gas) to monitor plants and trigger proper actions and notifications. The *"Soil Carer"* role represents the core activities of the *SFP*, its operation only makes sense in the context of the particular object and its contents (soil and plants). These activities depend completely on the local capabilities of this *SO* and the activities are only meaningful in the context of the soil the *SO* is taking care of, so this role needs to be played by the *SO* regardless of others playing it. On the other side, the *"User Mediator"* role and the other roles are *delegable* roles,

that can be played by another *SO* without affecting *SFP*'s main purpose.

In a collective of *SOs*, multiple *SOs* might play the same *delegable* role on purpose, for example, because the role activities depend highly in the context of the *SO* playing the role and multiple *SO*'s views are required for a plan. (E.g. temperature sensing from multiple points). However, there are also cases when roles are replicated just because the *SOs* are designed to work autonomously even if others are not available. In these situations, it is desirable that *SOs* stop offering a role which is broadly offered within the collective if that helps a particular *SO* to extend its availability. Depending on its internal or external context —E.g. battery level or physical space— an *SO* might decide to stop offering one of its *"delegable"* roles —and all the services linked to these— and so rely in the work of the other *SOs*. It might be because it has to save energy or because it has entered a space where there is a broad offer of the same roles by other *SOs*.

In practice, the *SO* Administrator determines which roles are *delegable* and defines the rules for triggering the management of them. This process could be carried out with a predefined frequency or once some context-based conditions are met. Even if the *SO* has a reduced number of offered roles, once the process is triggered, a criteria is needed to select between one role or other. We propose this criteria to be based on the estimation of both the partial role density and the role contribution.

Partial Role Density

Role density is a measurement that indicates how many *SOs* are available to play a required role within a collective. Calculating the role density having a complete view of the collective would be easy. However, each *SO* only has a partial view defined by the reduced list of the *SOs* it knows. This type of adaptation must avoid using excessive resources that could eliminate the benefit of switching off a role. Therefore, it is not worthwhile to query other *SOs* trying to obtain an updated and broad view of the collective. That would generate more network traffic, and additional processing for sending and receiving responses from others.

Instead, the *SO* has to rely in the list of known *SOs* to calculate a partial role density as follows:

$$prd_i = \frac{krs_i}{kso}, \tag{6.17}$$

where krs_i is the number of known *SOs* playing a role i within the collective and kso is the total number of known *SOs*. This measurement might be imprecise, but is a first filter that can be used together with another criteria. This measurement obtains cases where the density of roles is substantially high compared to the size of the collective, that these are detected even with a partial view. The more *SOs* are in the *"contacts list"* and the smaller the collective size, the more precise this measurement will be. However, it is not intended that *SOs* store every *SO* belonging to the collective, so the view will always be partial which, reinforce the need of complementing this measurement with another criteria.

SO's Role Contribution

In the case that role density do not provide The roles are uncoupled of the concrete workflows of operation —which are defined by plans and their activities—, these only identify the activities an *SO* is able to carry out. According to the triggered plans some activities of a role might be carried out while others no. The contribution is a measurement of the perceived importance of an *SO* to play a particular role, for the *SO*'s and collective's operation. This is an estimated measurement based only in the *SO* view and considering the execution profile of activities both individually and in cooperation with others. The rationale is that the most frequently the activities of a role are carried out by the *SO*, the most important is that *SO* for playing the role.

SIM monitors how frequent the activities are triggered within a set of context-related rules. These rules can define time windows E.g. (daily, weekly, etc) and physical places (E.g. meeting room, living room, etc.), among others. The individual *SO*'s contribution by role *rc* of a role i within a context x is defined as:

$$rc_{i,x} = \frac{qea_{i,x}}{toa_i} \cdot \sum_{j=1}^{toa_i} tie_{j,x}, \tag{6.18}$$

where $qea_{i,x}$ is the quantity of role i activities that have been executed within a context x over the total number of activities of the role toa_i, multiplied by the times all activities of the role i have been executed $tie_{j,x}$ within the time window.
In order to compare contribution of the *SO* for each available roles, we obtain a normalised role contribution $nrc_{i,x}$ with:

$$nrc_{i,x} = \frac{rc_{i,x}}{\sum_{i=1}^{r} rc_{i,x}}, \tag{6.19}$$

The roles with the minimum normalised contribution $nrc_{i,w}$ are the candidates to be disabled. In case of roles with the same *nrc* we use the minimum *tie*.
The Algorithm 4 presents the selection of roles for disabling.

```
input: The minimum role density parameter: rdParameter, the SO's
       contact list: contactList, the SO's delegable roles: myRoles,
       Count of triggered activities in the last time window:
       triggeredAct
1  for sobj in contactList do
2      quantityRoles ← countMyRoles(quantityRoles, sobj,
       myRoles) ;        // quantityRoles accumulates the
       quantity of SOs playing each of myRoles
3  end
4  for role in myRoles do
5      densityRoles [ role ] ←
       calculatePartialDensity(sizeOf (contactList),
       quantityRoles [ role ]) ;                          // 6.17
6      if densityRoles [ role ] > rdParameter then
7          candidates ← add(candidates, role)
8      else
9          contribution [ role ] ←
           calculateRoleContribution(role, triggeredAct,
           myRoles) ;                                     // 6.19
10     end
11 end
12 candidates ← add(candidates,
   minContribution(contribution));
13 for candidate in candidates do
14     disableRole( candidate);
15 end
16
```

Algorithm 4: Role Selection for Disabling

6.8 Summary

We have described our strategy for *SO* adaptation and presented the functions and methods that enable its implementation. The strategy is based in the dynamic selection of *SOs*, services and roles according to the context and the user preferences. At collective level, the *SOs* are able to select the *SOs* they cooperate with considering changing conditions. At individual level, *SOs* can decide every time they need to carry out an activity what is the service best suited to every situation. This approach is dynamic and flexible, it is driven by four group of factors: *trustworthiness*, *performance*, *efficiency* and *location*.

Based on these drivers, we defined a set of utility functions. These utilities enable to assess different candidates and so identify the best suited for a given context. Every time the context changes, these utilities are recalculated enabling to always obtain the best available candidates (*SOs*, services or roles). In the case of *SO* selection we have defined utility functions based on *reliability*, *Potential Performance* and *Estimated Distance To Target*. The data to calculate these utilities is gathered by the *em4so* middleware routines and is share through the *SO* protocol. For the service selection, we based the calculation of utilities from groups of *QoS* attributes matching the driving groups of factors.

Both *SO* and service selection were approached as multi objective optimisation problems. We introduced rankings defining user preferences that were transformed into weights to determine dominant factors within an aggregated utility function. The case of role selection was approached as a efficiency-driven optimisation. A utility function was defined including factors for a partial view of the role density and the role contribution for an *SO*.

Chapter 7

Evaluation

7.1 Introduction

In this chapter, we evaluate the *RbSO* software and *em4so* middleware architectures. We used several techniques in order to cover different parts of this work:

- ***Case Study***: This is aimed to evaluate C2, C3 and C4. This particularly covers the *RbSOs* conceptual model 4 and the *em4so* middleware architecture (Chapter 5), including the *SO* protocol and the adaptation method (Chapter 6), with a real testbed.
- ***Performance Evaluation***: This is aimed to evaluate C3. It addresses mainly the *em4so* middleware prototype implementation.
- ***Collective Evaluation***: It evaluates C2, C3 and C4, particularly the *SO* protocol (Chapter 5) and adaptation approach (Chapter 6), mainly in regards scalability and support to large number of *SOs*. We develop and use an agent-based model for simulation of *IoT* systems, showing the feasibility of C5.
- ***Qualitative Evaluation***: It assess from the perspective of the relevant *SO*b-*IoT* middleware solutions in regards the coverage of elemental processes and autonomy according to the foundations given in chapter 3. This also works as evaluation of C1, concretely the analysis tool that is used to assess other *SO*-based solutions.

For each part of the evaluation we present the design, results and discussion.

7.2 Research Challenges

The evaluation of the *IoT* systems is challenging as the standard techniques cover partial characteristics of these system at individual or network level E.g. [36]. On the other side, *IoT* test beds such as SmartSantander [85] provide a real infrastructure for operation and management of the cyber-physical devices with a defined architecture and approach for software development model, offering limited interaction over the base software of the involved devices. Multiple methods for *IoT* applications have been used highlighting particular aspects, but rarely, these enable the incorporation of heterogeneity, instability, large quantities of *IoT* devices in order to validate scalability and adaptation, at individual and collective level, of the middleware and specific *SO* software.

7.3 Contribution

We propose a novel agent-based model that enables the simulation of the *IoT* system operation covering mainly the collective behaviour and the characteristics of the *SOs* with larger number of *SOs*. This model enables simulation of multiple *SOs* that communicate and cooperate within a collective. On top of this model multiple metrics can be monitored and defined. We particularly define two metrics that enable the comparison of collective performance at the different situations under evaluation. To complete the evaluation of this work, this model is complemented with a case study working in a real setting and covering mainly individual aspects of the *SO* software development and the operation with a reduced number of *SOs*.

7.4 Case Study: Physical Resource Provisioning

7.4.1 Design

This part of the evaluation demonstrates the application of the *RbSOs* software development approach as well as the benefits of the *RbSOs* software and *em4so* architectures in the development of a real *SO*-based *IoT* application. The *em4so* architecture is evaluated through the prototype implementation described in section 5.10. Hence, the approach for the case study was to choose an *IoT* scenario and develop the application following the *RbSOs* approach and using the *em4so* middleware prototype. The case study is explanatory, focusing mainly in the individual operation of the *SOs* in order to

define goals, roles, scenarios, plans, capabilities activities and actions. The *SO* interactions are demonstrated along the few *SOs* available.

Particularly the case study highlights the following characteristics of the mentioned components of this work:

- Use of the *RbSOs* software architecture to build *IoT* applications with different heterogeneous participating *SOs*.
- Sufficiency of the *em4so* middleware architecture on-object components for a simple *IoT* application.
- Feasibility of the *em4so* middleware implementations to run on existing *SO* hardware platforms.
- Use of the *em4so* middleware to endow autonomy in regards user, platforms and other *SOs* to the *SO*.
- The operation of the *SO* communication protocol for cooperation and coordination at a low scale.
- The adaptation of an *SO* collective to unavailability on nodes, at low scale.

Three stages allow to observe these characteristics: the *SO* software engineering, the standard operation and the simulation of node/internet unavailability.

7.4.2 Scenario Description

We considered a case with common *IoT* requirements such as the support to heterogeneous capabilities, physical data gathering, adaptation based on the context and cooperation among others. Everyday objects, like home and office appliances, usually require consumables to operate. For example, a printer requires printing cartridges/toners, a vacuum cleaner requires filters, an air freshener a fragrance, etc. With the *SOs* in place, sensors detect promptly, when the physical resources get consumed and notify the operator, giving also information about where to get the supplies from (local stock or nearby supplier) or even triggering a purchase order.

Concretely, this case is applied to a smart home scenario where different *SOs* must play different roles in order to ensure provisioning of the physical resources. The scenario includes four *SOs*: an air freshener (*SAF*), a cleaning cupboard (*SCC*), a home Controller (*SHC*) and a TV (*STV*). *SHC* is simply a panel in charge of controlling the sensors around the home. The aim is to develop the *IoT* application and deploy it along the four *SOs* available, where the *SOs* are heterogeneous in regards the capabilities

Table 7.1: Case Study: Main files and properties per *SO*

	Roles	Goal	Observable Properties	Condition	Relevant Actions (services)
SAF	airFreshener	keep air fresh	room.activity	$= true$	spray
			saf_agent.freshener	$> 0\ ml$	
	pResourceConsumer	keep freshener deposit full		$\leq 1\ ml$	prepareRefillSpecs
					prepareUserNotification
SCC	lInventoryManager	keep control of inventory of cleaning products in home	scc_agent.demandRefill	$\neq null$	checkHomeAvailability
			home.freshener	$> 0\ units$	prepareUserNotification
				$= 0\ units$	lookUpNearbyStores
			store.details	$\neq null$	prepareUserNotification
			scc_agent.incomingRFID	$\neq null$	addProductToInventory
			scc_agent.outgoingRFID	$\neq null$	removeProductFromInventory
SHC	operatorNotifier	Notify user about *SOs* requests.	shc_agent.pendingNotification	$\neq null$	notifyUser
			msg_platform.pendingMessage	$\neq null$	processMessage
STV	operatorNotifier	Notify user about *SOs* requests.	stv_agent.pendingNotification	$\neq null$	notifyUser

they have. The hardware architecture of each *SO* is based on the Raspberry Pi Platform Model B and B+ with WiFi dongles. With Arch Linux 4.x as Operating system and Open JDK Zero VM 1.7.

The following is the expected behaviour for each *SO*:

- *SAF*
  - Detects movement and records it as observations.
  - When working as air freshener, sprays fragrance on movement detection.
  - Tracks the level of a simulated fragrance deposit.
  - Notifies the operator about the need of a refilling.
  - Since this *SO* does not have user interface it interacts with user through other *SOs* it has to discover.
  - Connects to the overlay network through the *SHC*.
- *SCC*
  - Tracks incoming and outgoing household items from the local inventory.
  - When there are not units of an item gather availability from known stores and notifies operator to repurchase.
  - When there are no stores registered it requests details about available stores to remote supplier system.
  - Since this *SO* does not have user interface it interacts with user through other *SOs* it has to discover.
- *SHC*
  - Obtain a list of nearby stores from cloud web services.

  - Send notifications via Telegram to the user.
  - Founder of the overlay network.

- *STV*
  - Display notifications to the user.

7.4.3 Stage 1: *SO* Software Engineering

The *IoT* application is made from the software components of each *SO* which is a set of:

- JSON documents including mainly properties of interests, activities, goals and scenarios.
- Service components implementing the capabilities of each *SO*.

The *SO*'s software was installed on top of the *em4so* middleware prototype in each *SO*. From the scenario specification we carried out design, implementation, testing and deployment of each of the *SO*'s capabilities and then we defined the required documents on top of these, according to the following process:

1. Identify the capabilities for each *SO* and develop the Service Components for each one.
   These are atomic services implemented in Java and are discovered via introspection by the *CM* which generates the JSON interface and stores it in the *KB*. For the developer, there is no need to define the interface of each service. Some examples of the implemented services are:

   - For *SAF* we implemented a service for configuring and reading from the motion sensor. This is a Java class that inherits from a middleware class *SensingService* and implements the method *public String readValue()*. This class configures the sensor using the Pi4J GPIO API[1] which is imported as a library, part of the *DF*. Within the *readValue* method the sensor is read returning a *boolean* value converted to *String*. It is a *String* as it supports different types of sensor readings. The *em4so* middleware takes this value and creates an observation that is transformed and stored as *knowledge item* by the *KSM*.

[1] http://pi4j.com/

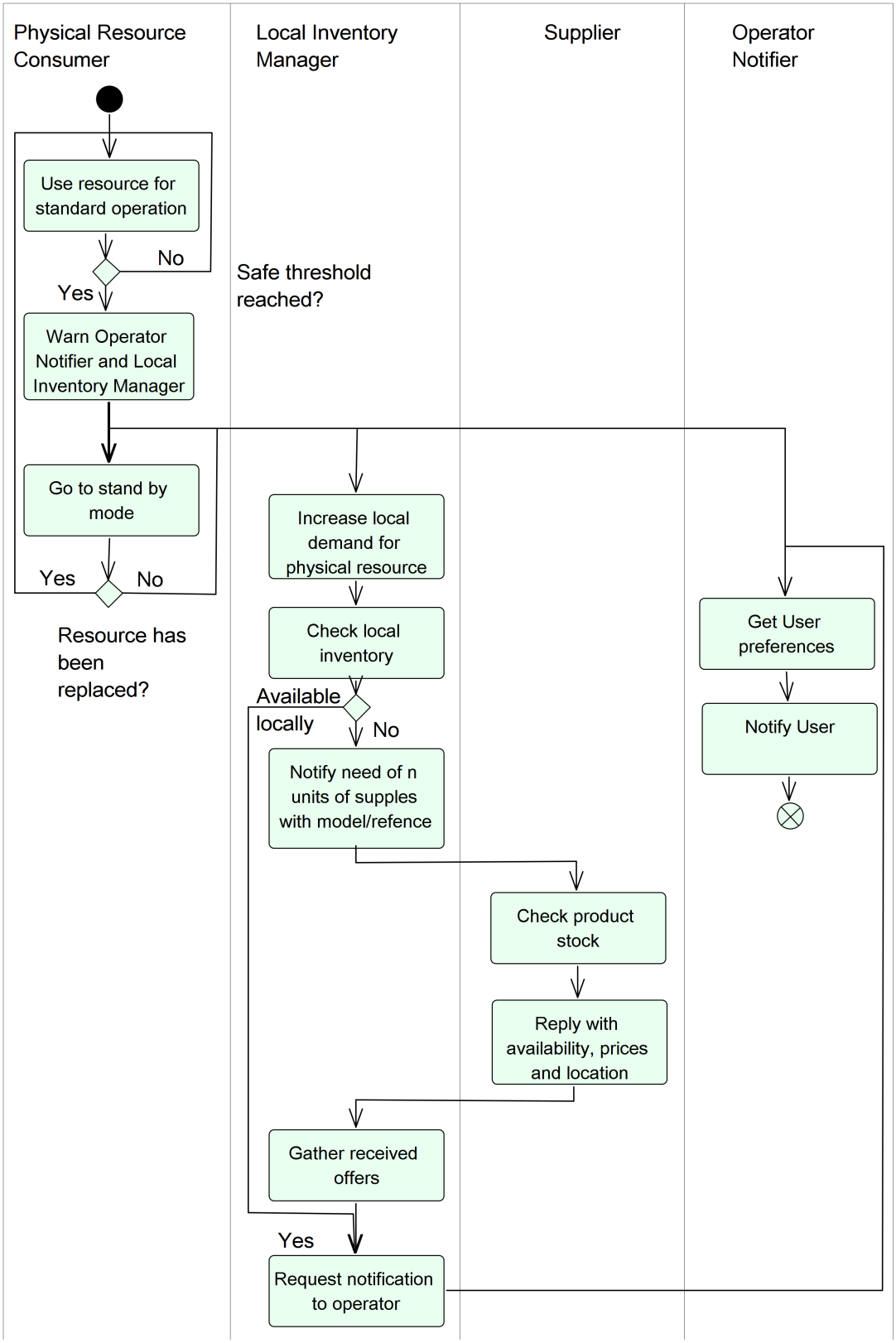


Figure 7.1: Scenario for Physical Resource Provisioning

- We implemented two variations of the service *notifyUser*. One in *SHC*, using Telegram Bot API[2], which connects to the Telegram Messaging platform and sends a message to the registered users. They receive these messages in their Telegram applications that is installed in the mobile phone or accessible via web browser. The other implementation in *STV* is simpler, it only displays a text message with the notification. These implementations are used to demonstrate how these actions are uncoupled from the activities enabling each *SO* to have its own version within the boundaries of a common activity definition.

2. Define the goals and properties of interest for each *SO*.
   We defined a hierarchy with one parent goal and two children. The parent goal runs forever —on progress state— and is required to group the other two parallel goals. The children goals are, one for standard operation, keeping a defined space fresh, and the other for resource provisioning, keeping fragrance deposit above a defined level. This latter *SAF*'s goal is a collective goal as require one *SO* capable to interact with users and another that provides information of the local inventory. Likewise we defined goals for the other *SOs* as presented in table 7.1. Since goals are defined as target states of the properties of interest, we needed to define these. The properties of interest for the physical resources include the usage level, the model/reference and the last date replaced. These are the *knowledge items* — defined according to the *RbSOs* knowledge representation 4.4.3— that are used in the activity definition.

3. Define the activities and roles that each *SO* can play/execute/monitor using the defined capabilities.
   We started with a general design of the activities required in each *SO* from the scenario and built the diagram presented in figure 7.1. Next, we define the activity documents containing the required: *knowledge items* , the triggering rules — simple conditions in terms of *knowledge items* that are processed by the *RE*—, a description, the output, the quality attributes, user preferences (optional) and the list of actions. These actions are named according to the services previously implemented and numbered indicating the flow of execution. An example of an activity document is found in the Appendix B.1. Finally, we defined the roles based on the case study description by grouping the related activities.

[2]https://core.telegram.org/bots/api

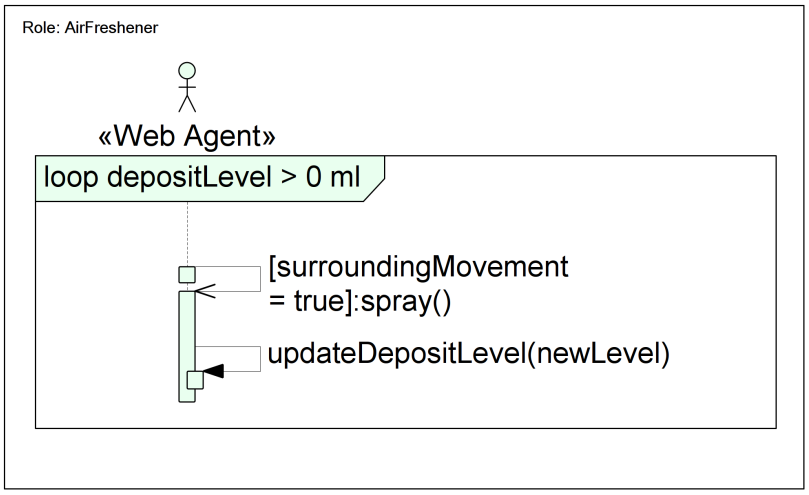


Figure 7.2: Case Study: Keep Air Fresh Scenario

4. Define the scenarios for each goal.
   According to the general design of interactions between roles defined previously, we specified the scenario documents containing the list of steps, in terms of activities, that are required for each goal. Besides the main collective goal we also define the scenario for individual goals, for example the one for *SAF*'s goal of "keeping air fresh" as shown in Figure 7.2. The first scenario ensures that fragrance is sprayed when surrounding movement is detected. A simple sequential scenario is presented in the Appendix B.2.

5. Configure *SO* and deploy.
   The configuration is made through definition of other auxiliary documents: a document for a known *SO* (to connect to the overlay), a document for the user and preferences. A text configuration file with mainly deployment paths, the *SO* id and transport protocol configuration. The deployment of the *IoT* application is done just by copying the jar files containing the Service components to the location configured and uploading the JSON documents to the *KB*. Note that at start up, each *SO* only has its own capabilities, roles, activities, properties, scenarios and the reference to another known *SO*.

7.4.4 Stage 2: *SO*-Based System Operation

In this stage we tested the standard operation of the *SO* collective. *SHC* as founder of the overlay, starts first and the others progressively. With the *STV* joining in the last

place. The *em4so* middleware ensures that every *SO* runs a web agent with an URL (i.e. IP + port + webcontext + agentId). The *em4so* middleware allows to enhance *SAF*'s limited capabilities by locating other *SOs* playing the roles needed, to achieve their defined goals.

Several processes happen at start up of every *SO*:

- Discover, load of each one's capabilities and generation of the interfaces in *SO*'s *KB*.
- Load goal hierarchy and start threads for monitoring pending ones.
- Start *SO* protocol for joining the network. The *TTL* defined is 1 as the number of participating *SOs* is low.
- Start threads for *SensingServices* defined that are linked to sensors monitoring properties of interest and recording observations in the *KB*.

We start with the *SAF* behaviour which is the one that triggers the main scenario. It starts monitoring two key properties of interest: one called *activity* from the *room* scope and the other *freshener* from *saf_agent* scope —the saf web agent—. We simulated the movement randomly and for the freshener, we assumed 1 spray spent approximately 0.1 *ml* of fragrance, and a refill is 250 *ml*. *SAF* loads the scenario and triggers the plan for the *keep fresh air* role. This keeps running until the *freshener* property indicates that the level of fragrance is below the specified in the activity *keepAirFresh* that contains the *spray* action. In this case, the activity cannot be triggered anymore until the deposit level is increased.

After observing a low level in the fragrance, *SAF* triggers the plan for physical resource provisioning from the defined scenario. It gathers the specification of the refill needed and then looks for the role in charge of the next activity: *checkHomeAvailability* which is part of the *lInventoryManager* role. It queries known *SOs* and finds *SCC*, to whom it transfers the *knowledgeItem* with the product refill specifications —the fragrance bottle model— and triggers the next activity. *SAF* also prepares the contents of a user notification, in this case indicating the deposit is empty. Then it detects that the next activity in the plan: *notifyUser* is responsibility of the *operatorNotifier* role, which it does not play. Hence, it queries known *SOs* and finds *SHC*, to whom it transfers the *knowledgeItem* with the notification details and triggers the next activity. *SAF* keeps waiting and monitoring *freshener* property to get back to the spray operation when the deposit is refilled.

SCC is monitoring the property *demandRefill* which is populated when it receives the refill specifications from other *SOs*. In this case, it receives one from *SAF*, so *SCC* end up triggering *checkHomeAvailibility* for the required product. In our first tests, we simulated enough units in the local inventory, so *SCC* triggers the next activity on the *SHC* to notify the user indicating that there is local stock. Then, when the local stocks is exhausted, *SCC* triggers the activity to look Up via simulated web service where to buy the product. *SCC* waits for information of the store and when received, it triggers user notification in *SHC*. *SHC* monitors pending notifications and sends them to the user via Telegram platform. *STV* has no participation in the standard operation.

7.4.5 Stage 3: Node/Internet unavailability

In this stage, we tested the autonomy of each *SO* from others by shutting down, during collective plan execution, one *SO*. We observed that, although activities of the unavailable role were, of course, not achieved, every other *SOs* were able to achieve their own activities. When *SCC*, playing the *lInventoryManager*, was not available, the *SAF* could still notify the user about the lack of consumables via *SHC*. When *SHC*, playing the *operatorNotifier* role, was not available, both *SAF* and *SCC* could still notify the user, when needed, via *STV*. In addition, once *SAF* had triggered *SCC*'s *checkHomeAvailability* and *SAF* went unavailable, *SCC* kept sending the user notifications —through *SHC*— and looking for the store details.

Likewise, when we dropped the internet connection, *SCC*'s store look up and *SHC*'s user notifications activities were not achieved but *SAF* and *SCC* could notify user via *STV*.

7.4.6 Discussion & Limitations

The *RbSOs* model and *em4so* middleware prototype were successful for implementing and executing the *IoT* scenario, showing that are a feasible solution for R3. The *em4so* was fundamental for easing and speeding this type of development. Compared to the traditional centralised approach where most of the application logic is implemented in a platform, in this case we had to develop four mini applications. Although every *SO* was based in the same hardware platform they were heterogeneous in the capabilities (sensing/actuating services) they hosted. The *RbSOs* abstractions were useful in defining each *SO*'s software individual structure and behaviour. Thanks to the *em4so* middleware, the development effort was focused in the concrete atomic functionalities

particular to each *SO* rather than in common tasks such as triggering goals, processing sensor data as input for activities, etc. The approach of capabilities realised as service components enabled us to share part of the functionality, for example when implementing the user notification services, between *SOs*, which was also a gain in effort and time. Each *SO* was able to discover others when joining the network. They successfully identified which roles each *SO* was able to play without the need of a central directory or platform having all the available roles.

We evaluated the autonomy of the *IoT* system as a whole and from the individual perspective of each *SO*. As a result, the *em4so* prototype demonstrated effectiveness in tackling R2. The *SOs* were able to complete the scenario without fundamental dependencies on third platforms. Each *SO* had control of both the application workflow and the data required to achieve a goal. Even when some nodes of the *SO* collective were not available the impact in the overall operation was minimised as each *SO* could still cooperate with the others available, showing this is a feasible solution for R4. A collective goal implies dependency, for example, *SAF* could not achieve the provisioning goal without *SCC* and either *SHC* or *STV*. However, this dependency comes from the lack of *SAF*'s capabilities which is a consequence of its hardware platform. In fact, if *SAF* had a screen or speaker it could play *operatorNotifier* role. So the *em4so* middleware is enabling the *SO* to extend its capabilities beyond its limitations, through others.

The *em4so* middleware applicability of course is not limited to this scenario. Additional roles and scenarios can be defined even using the same capabilities. For example, in the case of *SO* having movement detection sensor, the sensing service can be used for triggering an alarm, if it checks that it happened in a building during non working hours. Then, security staff can receive a notification in either a screen, a phone call or a voice message depending on where they are and the nearby *SOs*. In other situation, the same sensing service can be used just to activate the lights during a particular time.

The case was illustrative in the key advantages of the *RbSOs* model abstractions and the benefits of the *em4so* middleware in a real world scenario with a real hardware infrastructure. However, the case is limited mainly by the number of *SOs* and the complexity of the scenario. Considering the lack of availability of a real testbed with many *SOs*, we addressed scenarios with higher number of *SOs* via simulation in other parts of the evaluation. For more complex scenarios, we realised that more development and performance tuning was required for the *em4so* prototype. This was mainly because

in some runs of the case, we experienced delays of around 10-40 seconds, some times due to the sensor hardware and others due to the processing and transformation of the readings. These are implementation issues, that do not affect the overall approach and architecture, therefore we left them for a future work.

We found challenges when developing and managing applications that are distributed among various *SOs*. One of these is the maintenance and deployment of the versions of the different applications and middleware. This is a time consuming tasks and if done manually, is error prone because effective coordination of *SOs* require up to date versions of the middleware. Approaching the final of the development stage we found a relatively new tool[3], that deals with this problem as enable to develop and deploy applications in a single platform (PaaS) that later can be deployed at once to multiple device and updated from there. This is a recommended approach for developing applications on top of *em4so* middleware.

7.5 *em4so* Middleware Performance Evaluation

7.5.1 Design

The aim of this assessment is to evaluate the *em4so* middleware performance working over an existing real hardware architecture. This evaluation provides a view of how an *SO*, working with the *em4so* middleware, performs individually. Particularly, we determine how key components of the *em4so* architecture behave with different load units. This part of the evaluation is based on the prototype implementation and depends heavily in the programming language and software infrastructure used for it. The data was taken from a Raspberry Pi B+ using the Hyperic Sigar[4] for monitoring.

We focused on two key functions the capability/service loader and the *SO* Discovery. For the service load, we simulated loading up to 30 services. The simulated services where very basic, since the intention was to monitor the pure load process. For the *SO* discovery we simulated the creation/joining of an overlay network up to 20 nodes. The Raspberry Pi *SO* had to discover the other nodes that were started and run from an Intel Core i5 laptop.

[3] https://resin.io/blog/

[4] https://support.hyperic.com/display/SIGAR/Home

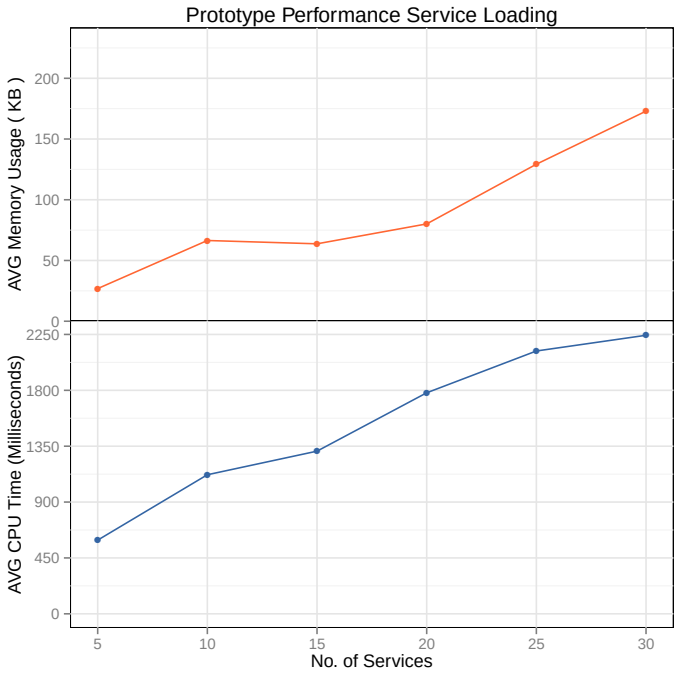


Figure 7.3: Performance Metrics Local Service Load

7.5.2 Results & Discussion

For the service/capability loader, the results are presented in figure 7.3. These show that the memory usage increases in the order of *KBs*, with an acute raise with more than 20 services. For the interval between 10 and 30 services the growth seems to be better described by a quadratic function. CPU usage starts in around 21 seconds which is roughly the time taking for *SO* booting, without triggering the service loader. When the number of services is increased the CPU usage rises at proportional pace. The difference between loading 5 services and 30 is around 1.6 seconds in CPU time. We observe that raise in CPU usage is sharper after the 15 services but then tends to get smoother after 25. We learned from this evaluation that the service loader implementation does not impose an excessive overload in the CPU. Therefore devices, with the hardware architecture under analysis, are able to load up to 30 basic services with less than 2 seconds of CPU time which indicates this is a load the device can handle properly. On the other side, the RAM usage grows fast while increasing the services and thus implies that, for the prototype implementation, the service loader consumes additional RAM resources when the services are increased. These results do not consider the particular resource demands from each individual service, that will vary according

to the type of service and this is expected to be higher when services access I/O devices.

For the *SO* overlay creation and *SO* discovery the results are presented in figure 7.4. The demands of memory for this function are in the order of *MBs*. For the CPU time, the difference between discovering 5 and 20 nodes was in the range of 8 seconds. These measurements give us an order of magnitude of the variations in performance when the load is increased. From the works reviewed we did not find similar metrics to compare with.

We observe that, compared to the service loading functionality, the *SO* discovery demands more CPU and RAM resources, although the increase, in relation to the quantity of *SOs*, is smooth and clearly linear. Even in the case of RAM it tends to stabilise faster as it grows. The results suggest that devices, with this hardware architecture and using the *em4so* middleware, are able to cope with the discovery of even higher number of *SOs*. One factor that is not analysed here is if the variable characteristics of the discovered *SOs* —E.g. roles offered— make any difference in the resource consumption when starting the overlay.

Our current prototype requires platforms with the Java VM support. Despite that everyday there are more *IoT* hardware platforms supporting JVMs[5], it imposes a constraint about the required resources for running the middleware. The *em4so* architecture, however does not hold that restriction and we left for future work to explore other middleware implementations (E.g. C and javascript/nodejs) with the idea of reducing the hardware requirements. It is also true, that our architecture is aimed at non-trivial objects, it is clear that in some scenarios with trivial objects it is not worthwhile to endow autonomy. E.g. a pen. Here, at most it would be desirable to store owners information and be able to be tracked if lost, probably also alert when ink is about to run out, but in this case autonomy for the purpose of the object is probably not worthwhile. Overall he *em4so* demonstrated to have a decent performance for a basic scenario which confirms it as a potential solution for R3.

7.6 Collective Evaluation

Considering the huge amount of devices required in order to set a proper test bed for *IoT* scenarios, this part of the evaluation was carried out through a simulation. The

[5] http://www.oracle.com/technetwork/java/embedded/overview/index.html

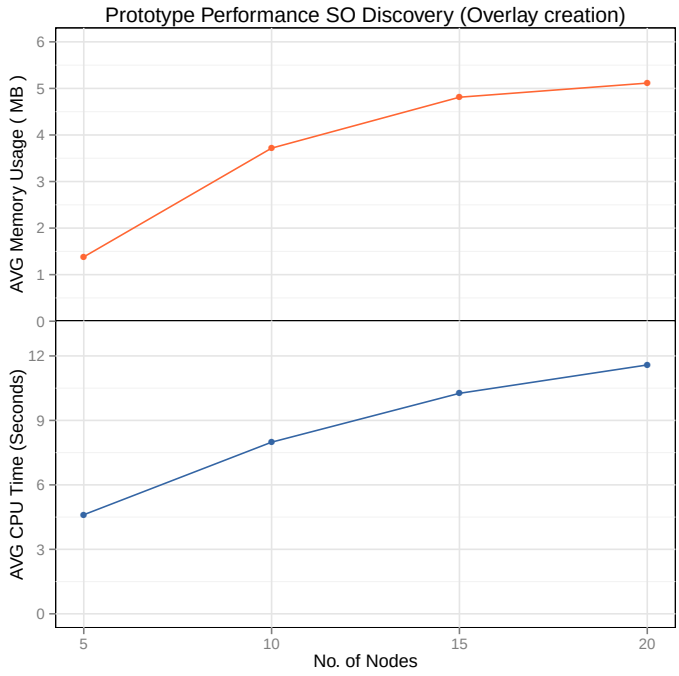


Figure 7.4: Performance Metrics *SO* Discovery

simulation approach is based mainly in the Agent-Based Modelling *(*ABM) and also incorporates elements from Montecarlo methods, essentially to include randomness to the *ABM* model.

7.6.1 Agent-based Modelling

In *ABM*, agents are though to work autonomously during a time frame which is measured in ticks. The equivalence of the tick, if required, is assigned for each individual model. *ABM* is particularly suitable for the evaluation of *IoT* systems for several reasons:

- It enables to setup highly heterogeneous scenarios. Experiments can consider several characteristics of the *IoT* devices, their environment and the other agents involved E.g. humans or third systems. These characteristics can be defined as parameters or variables in every particular scenario.
- It enables to simulate autonomous entities as the *SOs* are.
- It enables to monitor characteristics and behaviour at micro and macro level.
- It enables to simulate highly volatile situations.

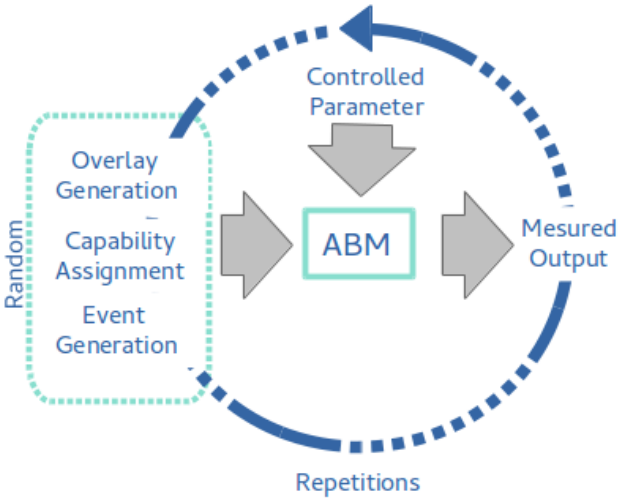


Figure 7.5: Collective Experiment Design

7.6.2 Design

The main goal of this evaluation is to resemble operation of cyber-physical systems — i.e. the *SO* collectives— and particularly, highlight the following key features of *em4so* middleware:

- Scalability of the *em4so* middleware approach and *SO* protocol to different conditions.
- The adaptation of an *SO* collective, at a high scale of expected *SOs*.

The design of the experiments of this section is presented in figure 7.5. The core element is an *ABM* that represent the *IoT* system deployed within an environment. This environment is a physical setting, it could be a residential building, a factory, a school or a neighbourhood. Agents are essentially the *SOs* that interact in order to achieve defined goals. The model was implemented in the Repast[6] agent-based simulation platform. The links between *SOs* are the overlay network connections built from the *em4so* protocol. Each *SO* has installed a *em4so* middleware prototype.

The model is feed by a set of random-generated data covering:

- Order of the *SOs* joining the overlay and the time to do so.
- Capabilities assigned to each *SO* following a given distribution.
- Generation of events that include triggering of plans, departure and rejoining of *SOs*.

[6]https://repast.github.io/index.html

There is also a set of controlled parameters that are part of the *SO* characteristics defined below. Given these inputs we run the model for a number of repetitions and obtain the metrics of interest according to each particular experiment. For the execution of these simulation cycles we used the ALICE[7] High Performance Computing Facility at the University of Leicester.

Model Parameters

SOs are the autonomous and heterogeneous entities that have installed an *em4so* middleware implementation and ad hoc application software. The heterogeneity supported in the model is defined by the hardware characteristics and the particular mix of capabilities and roles every *SO* plays. Each *SO* has a set of deployed services that work atop the existing hardware capabilities. The mix of capabilities simulates the different types of sensors, actuator or processing functionalities the *SO* might have. Roles are assigned to each *SO* based on their capabilities. Each *SO*'s *KB* includes the roles they can play and the scenarios for achieving the goals they have.

Likewise, the *SO* collective exhibits some characteristics that are consequence of the characteristics of the *SOs* that belong to it. The *Service density* indicates how many *SOs* are offering a particular service. As the services enable the execution of the activities this can also be regarded as *Activity density*, meaning how many *SOs* are able to carry out an activity (See assumptions 7.6.3). Since roles group a number of activities, the density of activities is proportionally linked to the *Role Density*. The more spread a role throughout a collective, the easiest to find a suitable player for that role within the collective.

The table 7.2 presents the most relevant characteristics of the *SO* and the collective of *SOs* that are defined as parameters within the model. The scope of the parameter indicates how the parameter is defined. For example, the four hardware configuration types are defined at model level, but how many *SOs* are of each type is defined per experiment.

[7] http://www2.le.ac.uk/offices/itservices/ithelp/services/hpc/alice/about

Table 7.2: Main Simulation Parameters

Name	Description	Scope	Values	Units
Hardware				
Processing Power	Maximum amount (millions) of instructions the *SO* is able to process in a time unit (tick).	Model	1	Mipt
Config type	A combination of: No. of Cores, RAM and Storage, respectively.	Model	A (1, 0.5, 2) B (2, 1, 16) C (4, 2, 32) D (4, 4, 64)	Cores: Units RAM: Gb Storage: Gb
SO per type	Percentage of *SOs* per Config type	Experiment	[1-100]	%
Battery	Battery powered	*SO* type	[0, 1]	(Discrete)
Network				
TTL	Time-to-live for messages	Model	4	hops
PING Frequency	How often *SOs* PING others	Model	30	ticks
em4so				
Roles	Roles played by the *SO*	*SO* type	[1 -7]	(Discrete)
Role:Activity Proportion	Proportion of Activities per Role	Experiment	1:1 & 1:5	
Service/ Activity/Role Density	Percentage of *SOs* offering a service/able to carry out an activity /play a role.	Experiment	[1 - 100]	%
Max Known *SOs*	Maximum number of *SOs* to be stored in contact list.	*SO* type	[4 -7]	(Discrete)
Timeout	Maximum time an *SO* waits to find the next player before quitting the plan.	Model	30	ticks

System Behaviour

SOs are able to send and receive messages following *em4so*'s *SO* protocol. Each *SO* schedules the work it carries out in every time unit considering the available hardware resources. See assumptions in section 7.6.3. The work includes planned activities as well as tasks for adaptation and passing and processing the different protocol messages. Tasks accepted to be processed by the *SO* within one tick, are put in its queue to be processed from the next tick. The amount of instructions of each task is split to fit in the available processing capacity for the *SO* and the tick.

We defined two metrics to measure the performance of the *SO* collective: *Mean Query Time* (*MQT*) and *Plan Success Rate* (*PSR*). *MQT* is calculated for the plans that the *SO* collective completes and is an indicator of how quickly the *SOs* are able to locate other cooperating *SOs* within the collective. *MQT* is a component of the total execution of a plan, the rest of the time depends on the power and workload of each *SO*. The *MQT* is calculated:

$$MQT = \frac{\sum_{i=1}^{n} TQTP_i}{n}, \tag{7.1}$$

where $TQTP_i$ is the total query time for a plan i of n plans completed by the collective during the period of analysis. *TQTP* is calculated as the sum of the query time of every activity of the plan.

On the other side, *Plan Success Rate* is calculated as follows:

$$PSR = \frac{CP}{TP}, \tag{7.2}$$

where *CP* is the quantity of completed plans and *TP* is the quantity of triggered plans within the collective during the period of analysis. These metrics are analysed during the experiments in order to determine how the system performance is affected.

We define the following common characteristics for the system's behaviour along the experiments:

- The system is evaluated by triggering plans with activities of different characteristics.
- The system workload is defined by the quantity of plans to be executed.
- The plan requires the *SOs* to query for others in order to continue carrying out the activities of the plan they are not able to perform.

- Activities are performed by different *SOs* of the system.
- There are five scenarios with seven steps each one requiring the mentioned activities.
- The *SOs* have different hardware configurations and they spent their resources carrying out these activities but also sending and replying messages and deciding about ongoing plans.
- The *SO* collectives are evenly heterogeneous with 25% of the *SOs* of each type in every collective.

7.6.3 Model assumptions and limitations

Assumptions

- For simplicity, each activity includes one single action which is carried out by one service. For this reason, for the model, the service density is equal to the activity density. Actions are managed as tasks to be processed by an *SO*.
- Besides actions, which are specific for each *SO*, relevant tasks include also common routines for adaptation and message passing and processing. Other tasks are assumed to produce an even load in the *SOs* and therefore these are not considered.
- Tasks have a size which indicates the set of fixed instructions required to process.
- The speed of processing a task by an *SO*, depends on the task size as well as on the *SO*'s configuration type. The more powerful the *SO*'s configuration type and the lower the size of the task, the quicker the *SO* is able to process it.
- Before joining every *SO* has the address of another *SO* that is already in the network.
- Battery and storage usage patterns are independent of the concrete tasks being processed. E.g. battery usage might depend on the hardware characteristics of the *SOs* such as the presence of a screen; likewise, storage might depend on the data to store which might vary for different executions of the same activity by the same *SO*.
- The exit condition of the experiments is the completion of the triggered plans or 1200 ticks.

Limitations

- Hardware resources —particularly CPU, ram, storage and battery— and their scheduling / allocation / usage patterns were modelled as a consistent simplifi-

cation of those existing in real world. The reasons are: (1) The aim was to incorporate *SO* heterogeneity at this level and their impact in each one's behaviour during operation, rather than a strict modelling of each resource operation. (2) Real world patterns are complex considering multiple factors that go beyond the scope of this work.

- Due to the absence of real world data for the *SOs* and the collective, the data was generated following real world constraints E.g. hardware power available for *IoT* devices. We tuned the parameters having as reference the results of the prototype performance and adjusting after several cycles of evaluation.

7.6.4 Experiment EX1: Scalability Evaluation

The purpose is to assess the performance of the *em4so* middleware, increasing the quantity of participating *SOs* and the workload of the system. Besides the common characteristics, for this experiment the heterogeneity in offered services is incorporated with three types of activities with different density: *High density* in 30%, *Medium density* in 20% and *Low density* in 5% of the *SOs* of the collective, respectively.

Test case 1: Increasing number of *SOs*

This case simulates a situation where the system is composed by different quantities of *SOs* (*nSO*) and a stable workload of 10 plans is randomly triggered. The range evaluated for *nSO* is from 20 to 140, which covers reasonable expectations for the overlay networks under analysis.

Results

Figure 7.6 shows the *MQT* and *PSR* results for 50 runs with two role/activity proportions. Each run represents a different collective with the corresponding number of *SOs*, so for each *nSO* value there are 50 different collectives. In figure 7.6a, each shape represents a role/activity proportion, the triangles are for values obtained with five activities per each role and the circles are for the ones obtained for one activity per role. A slightly bigger triangle or circle shows the mean value for the 50 runs along each quantity of *SOs*.

The first observation is that there is a proportional increment in the *MQT* values when *nSO* increases. The displayed growth is close to a linear function, with a more acute

slope for values of $nSO \leqslant 60$. This shows that the greater the *nSO*, the growth tends to stabilise, which makes sense as there are more *SOs* available offering the roles and activities and then it is quicker to find them, therefore reducing *MQT*. The box plot on the right of the figure 7.6a shows that *MQT* values are consistently distributed with a difference lower than around 7 ticks throughout the different *nSOs*, perhaps slightly increasing for greater *nSOs* and proportion *1-to-5*. This low variability of the data obtained strengthen the observations of the *MQT* growth along the *nSO*.

In regards *PSR*, we plotted the frequency polygons for the 50 runs with the two proportions indicated. Figure 7.6b shows that the most of the *PSR* values are close to 1.0 and only a few $\leqslant 0.9$. Particularly, only when $nSO = 40$ and $nSO = 20$, there are a few cases when $PSR \leqslant 0.6$. Since it is consistent for the different *nSO* values and role/activity proportions, the *PSR* shows that the system is stable in completing the plans, regardless of the *nSO* values.

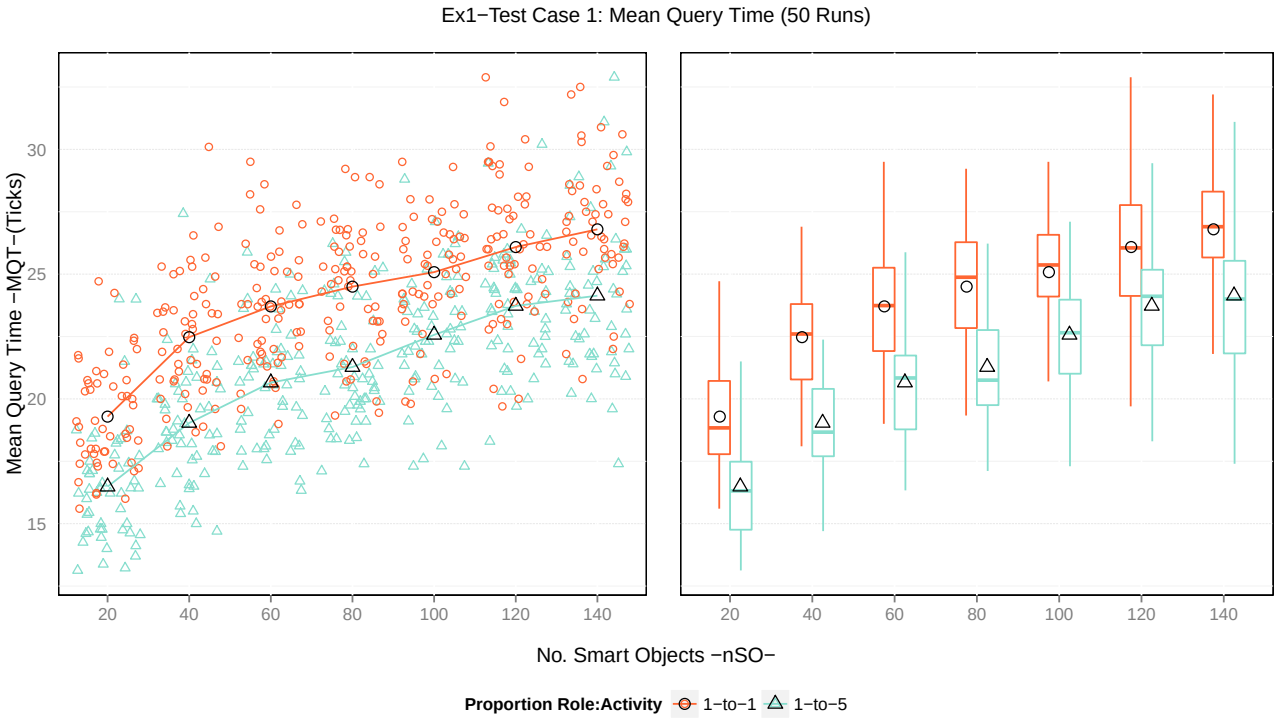
The role/activity proportion enables to determine the difference, if any, between working with roles (*1-to-5*) or with activities —services— at individual level (*1-to-1*). Indeed, we appreciate a consistent difference the mean values of around 5 ticks for the *MQT* values, in favour of having roles. This is explained as each *SO* has a limited space for storing other *SOs* activities/roles. When the activities are grouped they gather more information of the abilities of each other than individually. Considering that by default, *SOs* have a role definition that indicates which activities are associated to each role.

Test case 2: Increasing Workload

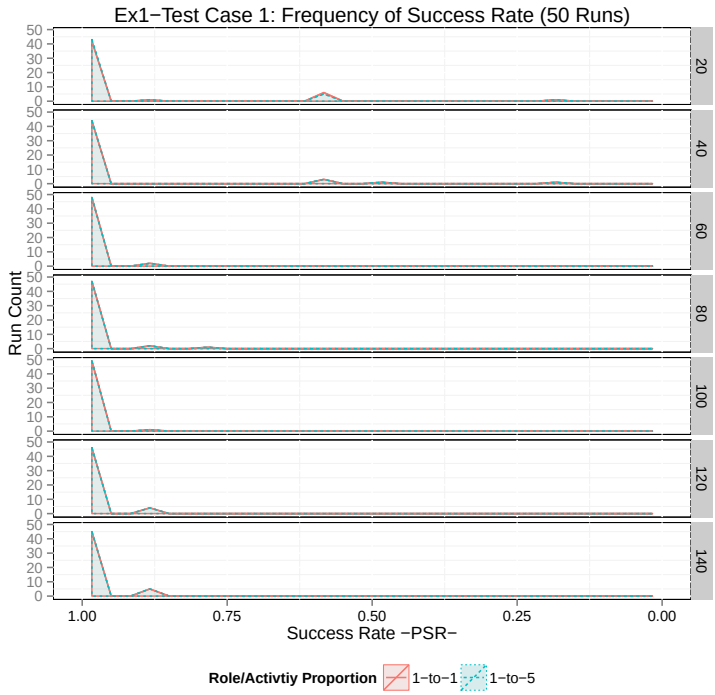
This case simulates a situation where the system is composed by a stable *nSO* and an increasing workload of plans $nPlan$ is randomly triggered. The range evaluated for $nPlan$ is from 10 to 90 with a $nSO = 40$.

Results

Figure 7.7 shows the *MQT* and *PSR* results for 50 runs with two role/activity proportions. We observe an inverse relationship between the $nPlan$ and the *MQT*. A combined analysis of *MQT* and *PSR* shows that the collective keeps stability in completing the triggered plans and is able to reduce the *MQT* while the workload is increased. This is explained because *SOs* store in their "contact lists" the *SOs* they have cooperated with and their roles, enabling them to complete further executions of plans involving

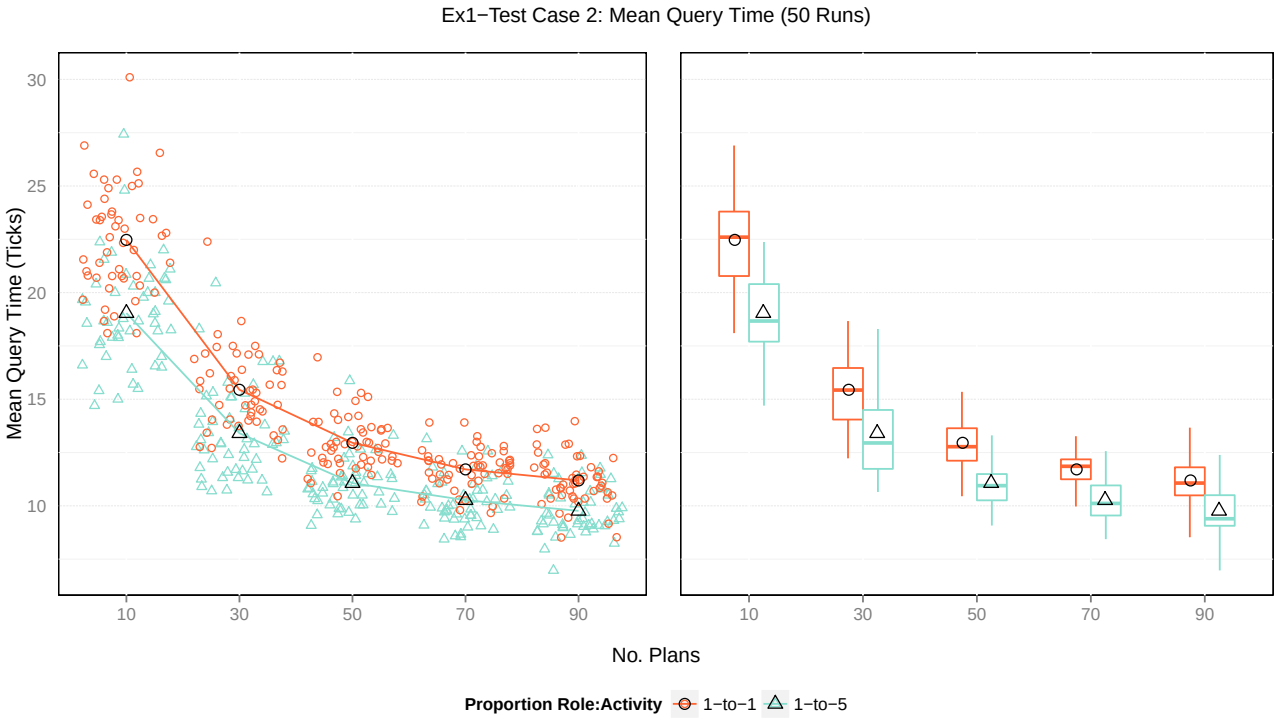


(a) Mean Query Time

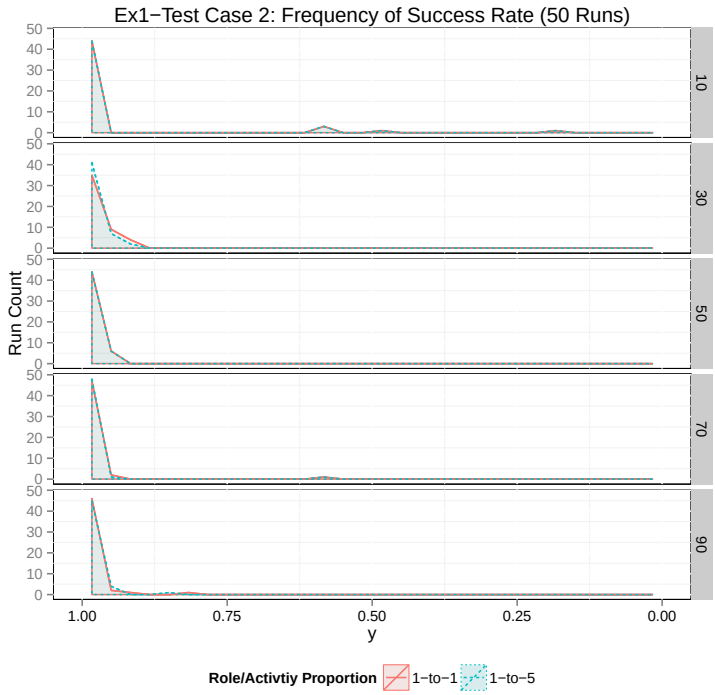


(b) Success Rate

Figure 7.6: *em4so* middleware Scalability Results: Increasing No. of *SOs*



(a) Mean Query Time



(b) Success Rate

Figure 7.7: *em4so* middleware Scalability Results: Increasing No. of Plans

the same roles, without the need of additional queries. Although plans are triggered randomly, a successful cooperation between two *SOs* is reinforced by the reliability criteria considered in the *SO* selection. The box plot in this case shows data with even lower variability than the one in the test case 1.

In regards the role/activity proportion, analysing the mean values, there is a noticeable difference in favour of having roles. However, in this case this difference decreases slightly from a value around 7 ticks to a value around 3 ticks when the workload increases. This is due to the same reason previously described, after the initial cooperation, the "contact list" cache represents and advantage for allocating other *SOs*.

7.6.5 Experiment EX2: System Adaptation Evaluation

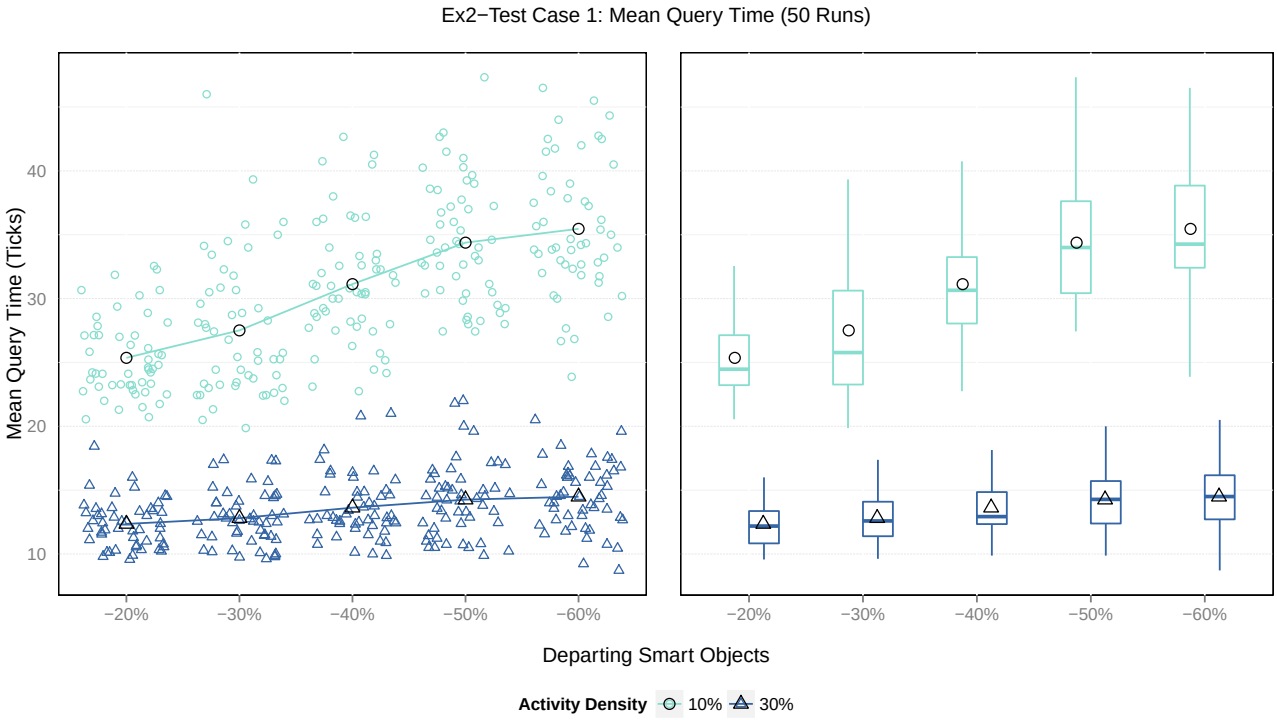
The main purpose is to evaluate how the system adapts to unstable conditions, particularly the departure and joining of new *SOs*, with different densities of activities and while executing a set of triggered plans. At the same time, this case is designed to demonstrated that, thanks to *em4so* middleware, *SOs* within the collective are autonomous from each others being able to carry on with their tasks even if some of the *SOs* are not available. In this experiment, we also compare how the different densities of activities have impact in the adaptability of the system. For this purpose, we work with densities of 10% and 30% of *SOs* offering each activity.

Test case 1: *SOs* Departing

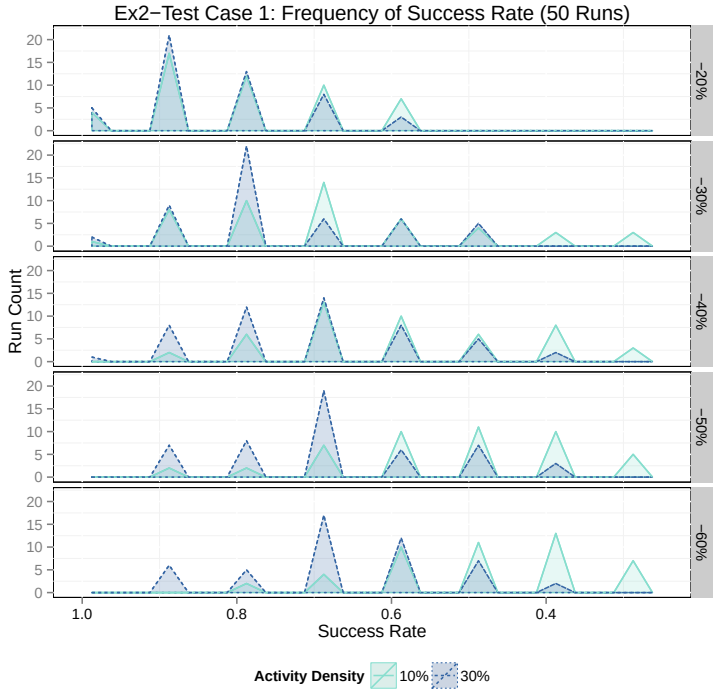
This case simulates a situation of different numbers of *SOs* departing from the collective. The collective starts with $nSO = 40$ and a stable workload of plans is randomly triggered. During the execution of these plans a range between 20% and 60% of the *SOs* departs randomly and progressively from the collective.

Results

Analysing the *PSR* frequencies presented in figure 7.8b, we observe that departures cause the *PSR* to be distributed between 0.3 and 0.9. There is a shift in the frequency distribution, from the acute peaks in the higher values of *PSR* when the percentage of departures is just 20%, to the more lower flattened peaks when the departures are of 60%. However, the figure shows that for density of activity = 30% and the departures $\geqslant$ 40%, the *PSR* is relatively stable, showing a very similar frequency distribution. For fewer departures, the performance is better with most of the *PSR* values between 0.8



(a) Mean Query Time



(b) Success Rate

Figure 7.8: *em4so* middleware Adaptation Results: Departing *SOs*

and 0.9. On the other side, when the activity density = 10%, the system is more sensitive to departures and then the frequency of lower *PSR* values increases clearly.

Observing the *MQT* presented in figure 7.8a there are two very different situations according to the activity density. On one side, when the density is 30%, the departures of the *SOs* are almost unnoticeable keeping a smooth *MQT*. On the other side, when density is 10%, not only the plans completed are fewer with more departures but also the *MQT* of the completed ones is higher. The box plot shows a more homogeneous *MQT* value distribution for higher density of activities. These finding together with the previously seen stable *PSR* values show that the system adapts to the conditions of having multiple service provider available (density = 30%) and takes advantage of them to carrying on with the triggered plans, causing the minimum variation in performance given by *MQT* and *PSR*.

Test case 2: *SOs* Rejoining

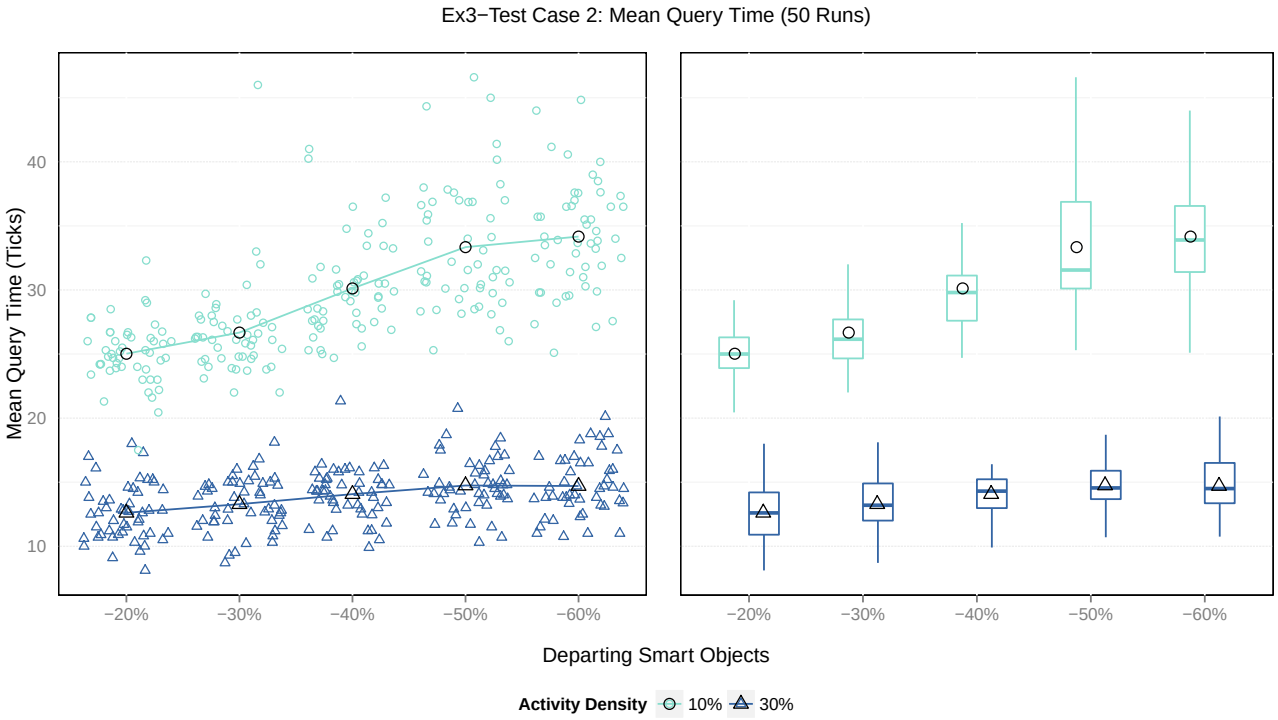
This case is complementary to the previous one and is designed to show how the system is able to incorporate new *SOs* in the execution of the triggered plans and recover from caused disruptions. For this case, the *SOs* that had previously departed from the collective, rejoin after 30 ticks.

Results

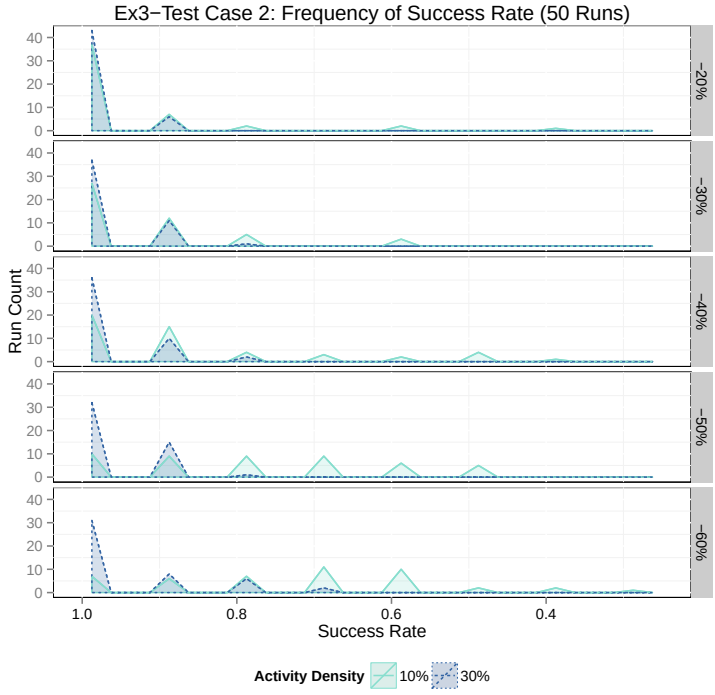
Figure 7.9 presents the results for the rejoining of the *SOs*. Compared to the test case 1, the results are almost identical with minimal improvements in the mean for the *MQT* shown by the slight variation in the slope of the lines. However, there is a significant improvement in the *PSR*, particularly for density = 30%, where we observe that most of the cases have *PSR* values $\geqslant 0.8$. For density = 10% there are also improvements as most of the cases are distributed around *PSR* values $\geqslant 0.5$. Again, these results show the ability of the system to adapt to the new conditions where new returning *SOs* are available which has impact in the increase of completed plans.

7.6.6 Discussion

The results of the agent-based simulation have shown that *em4so* middleware scales lineally to the increasing number of *SOs* and is able to take advantage of previous interactions with other *SOs* in an scenario of a high workload. Likewise, the system is able to adapt to the changes related to the outgoing and incoming *SOs* and incorporate them



(a) Mean Query Time



(b) Success Rate

Figure 7.9: *em4so* middleware Adaptation Results: Rejoining *SOs*

to the collective operation. All of these, considering a scenario with heterogeneous *SOs* in both their hardware platform and the capabilities offered. These results confirm case study findings on effectiveness of our solution for tackling R2 and R4. In this case, by scaling the workload and number of *SOs*, part of the collective.

We have learned from this evaluation that performance of these *SO* collectives depend on multiple variables. We observed that changes in the activity density and the proportion of roles/activity have impact in the performance of the system. Therefore, the perceived advantages of the *em4so* middleware depend also in the configuration of the collective.

7.7 Qualitative Evaluation

7.7.1 Design

Using the *E-Ma-Gen3* analysis tool proposed in section 3.5 we carried out a qualitative comparison of the main middleware solutions presented in the chapter 2.4. We focused in the support that each middleware solution gives to the set of fundamental processes along the key areas defined in *E-Ma-Gen3*. Hence, we analysed two perspectives. The first one is the middleware support to the *SO*'s fundamental process within the elements of an area. This support can be offered in many ways, for example: routines, abstractions, API or runtime services. If the solution offers any of these, we indicate then the process is supported. The second perspective indicates if, the support offered, enables the *SO* to keep autonomy relative to human users and exogenous platforms, or if, on the contrary, it imposes any dependency from the user or an exogenous platform.

7.7.2 Results & Discussion

The results of the comparison are presented in table 7.3 below we summarise the main findings:

- Analysing autonomy across the different processes and areas, we observed that only Leppänen, *em4so* and partially *ASAWoO* consider the autonomy from exogenous platforms. Only *em4so* provides a solutions that is not based on the concept of *Central Directory* and give the basis for a communication, coordination and cooperation based on a P2P *SO* protocol. Leppänen and *ASAWoO* solutions consider the existence of the *Central Directory* where *SOs* register their resources

Table 7.3: *IoT* Middleware Architecture Comparison

		Knowledge			Behaviour			Resources			Relationships			Structure		
		Generation	Management	Exploitation	Generation	Management	Exploitation	Generation	Management	Exploitation	Generation	Management	Exploitation	Generation	Management	Exploitation
UbiWare	HrA	●	○	●	●	○	●	●	●	●	○	○	●	●	●	●
	PrA	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
FedNet	HrA	–	–	–	–	○	○	●	●	●	–	–	●	●	●	●
	PrA	–	–	–	–	○	○	○	○	○	–	–	○	○	○	○
ACOSO	HrA	●	●	●	○	●	●	○	●	●	○	●	●	–	○	●
	PrA	○	○	○	○	○	○	○	○	○	○	○	○	–	○	○
ASAWoO	HrA	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●
	PrA	○	○	●	○	●	●	●	●	●	○	○	○	○	●	●
IoTSilo	HrA	–	–	–	–	○	○	●	●	●	–	○	●	○	●	●
	PrA	–	–	–	–	○	○	○	○	○	–	○	○	○	○	○
Leppänen	HrA	–	○	●	–	●	●	●	○	●	–	–	●	●	●	●
	PrA	–	●	●	–	●	●	●	○	●	–	–	●	●	●	●
em4so	HrA	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	PrA	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

HrA: Human-relative Autonomy, PrA: Exogenous Platform-relative Autonomy

●: Solution supports a process and gives relevance to the *SO*'s relative autonomy,

○: Solution supports a process but does not give relevance to the *SO*'s relative autonomy,

–: Solution does not indicate support to the process

y/o services, however the former one indicates that this could implemented as P2P although do not give details [74]; on the other side, *ASAWoO* requires a *Functionality Directory* to determine what avatars can do [79], this is required to determine how to execute a particular application, however part of the individual functions of the *SO*, mainly *OSGi*-based, do not require interaction with others. The other solutions rely in the *Directory Facilitator* of the agent platform or in a Message bus.

- In regards knowledge, we observed support to decision-making functions in most of the solutions except *FedNet* and *IoTSilo*. The management of the *SO*'s knowledge repository is only tackled in *ACOSO*, *ASAWoO* and *em4so*. In Leppänen it is considered as part of a broader mobile agent repository and is regarded as a standard database [74]. *ASAWoO* is strong in the use of semantics for annotation of functionalities, reasoning and decision-making. In general, there is a lack of combined techniques for knowledge generation, including not only reasoning but also machine learning and deep learning approaches.

- *ASAWoO*, *ACOSO UbiWare* and *em4so* stand out among the others in the processes related to the behaviour. These provide good abstractions that enable to manage and use distinctively different *SO* capabilities and to adapt the *SO* behaviour to the context. *Avatars*, *User-Defined Tasks*, *Reusable Atomic Behaviours* and Roles/Capabilities, respectively, are the key pillars of the *SO* behaviour in each solution.

- Data generation and exploitation without human intervention is a must in *SO*-based middleware and so it is widely supported by the compared solutions. *UbiWare* and *IoTSilo* have a clear orientation towards resource management and offer simple abstractions for configuration and use. In regards social resources, only *em4so* makes use of reputation for decision-making and operation within a collective of *SOs*.

- We focus in the cooperation and social relationships that might require more support from the middleware solution. Cooperation of *SOs* is addressed specially in *ACOSO*, *ASAWoO* and *em4so*. *em4so* stands out in the approach for relationship generation and management, in most of the other solutions, relationships with other *SOs* are considered mainly as a specific transient communication. *em4so* is the only one that considers long-term relationships and use the information

provided from previous interactions with other *SOs* in order to make decisions of new cooperation.

- Fundamental processes along the *SO* structure are widely supported. Leppänen, *ASAWoO* and *em4so* stand out as tackle specifically structural adaptation. The former solution use mobile agents which are composed dynamically via Web service interfaces. The latter solutions use implementation of the service components, in the case of *ASAWoO* is more robust as is based in the *OSGi* framework, however this also makes it heavier than *em4so*. *em4so* has a minimal implementation and service interfaces are generated as documents that have direct transformation into knowledge items making them available for reasoning process with few required processing.

From the surveyed solutions, these results show that *em4so* middleware architecture is the only one addressing relative autonomy as a central characteristic, then contributing to fill the gaps identified in R2 and R3.

7.7.3 Threats to Validity

We identify the main threats to the validity of our study considering each part of the evaluation:

- **Case Study**
  Given that the size of the sample, in this case the number of *SOs* that we use for the study, is not characteristic of a typical *IoT* scenario and the simplicity of the executed scenario. We addressed this threat with a set of complementary experiments using the agent-based simulation.

- **Middleware Performance Evaluation**
  The results obtained in this evaluation show conclusions that can only be applied for the particular implementation. There are different variables that influence the performance of the solution including the implementation of the Java Virtual Machine, the version of the API, the use of the framework for monitoring and gathering operating system and hardware-level information and the particular programming of the reference architecture.

- **Agent-based simulation**
  Results obtained can be affected by the particular agent model that we employed. Although, we incorporated as much as possible of the properties of individual

SOs and collectives, our model reflects only a limited set properties of a real-world scenario (See section 7.6.2).
Besides, the model and initial parameters required some input data for generation of overlays, event triggering, hardware and network properties. Due to the lack of data available following the *SO*-based development, the values were based in our results obtained in the case study and the performance evaluation of the prototype. In order to reduce this threat, the input data was generated randomly and the parameters tuned after running several simulation cycles. One tuning that is pending is to gather the *Mean Query Time* and *Plan Success Rate* measurements from a real-world test bed, using our prototype in small or medium scale scenario.

- **Qualitative Evaluation**
  The tool we employed for comparison of the existing solutions is our own proposal, making the evaluation potentially biased. First, because the tool is potentially incomplete (See section 3.5) and second because the it makes emphasis in the elements we focused our study on, including relative autonomy as well as the fundamental processes and key areas identified for the *SO* systems. However, due to the lack of other tools that include this view of the solutions, our proposed tool was the best alternative available for a qualitative evaluation.

7.8 Summary

This chapter has presented the evaluation of the *RbSO* software architecture and the *em4so* middleware architecture including its adaptation features. We have utilised different approaches for validation including definition of a case study, performance evaluation of the *em4so* prototype, agent-based simulation and qualitative comparison.

We developed a case study based on a physical resource provisioning scenario. We used the implemented prototype based on the *em4so* architecture and developed a distributed *SO*b*IoT* application. The application was based on the *RbSOs* approach involving the specification of goals, scenarios, roles, activities, capabilities and properties of interest for each involved *SO*. The case showed the advantage of the *em4so* architecture in developing software for devices that have control of the application workflow, the data generated, are able to reason and determine which goals to pursuit and cooperate with other *SOs* in achieving collective goals. These *SOs* can operate even when there is no

connection to a remote platform.

With the *em4so* middleware prototype we carried out a performance evaluation focused in the *SO*'s functionalities for the *SO* discovery (overlay creation) and the service/capability loader. The results show that these key functions run properly in devices with a constrained hardware architecture as the Raspberry Pi B+ used. However, the service loader functionality imposes an overload in the RAM resource usage, without considering the specific load of the services. On the side, the *SO* discovery results show that this combination of hardware architecture and *em4so* middleware is able to cope with increasing number of *SOs* with small variations in CPU and RAM demands.

The agent-based simulation enabled the evaluation of *em4so* middleware at a collective level. To compare performance with different conditions we defined two metrics: the *Mean Query Time* and the *Plan Success Rate*. From the scalability results we observed that the performance a collective working with *em4so* middleware, based in its *MQT*, scales linearly as the number of *SOs* increase while the *Plan Success Rate* is stable throughout. In addition, with *em4so* middleware the *SO* collectives use *SO* details from previous interactions and hence the *MQT* decreases while the *Plan Success Rate* is kept stable. We observed that *em4so* middleware makes possible the adaptation of *SO* collectives by taking advantage of the available *SOs* for completing their plans, rather than be stuck to specific *SOs*. The *MQT* variations are even minimal when the activity density is 30% regardless of the quantity of *SOs* departing. The *em4so* middleware also makes the *SO* collective to adapt to new entrant *SOs* by using them to stabilise the *PSR* in values greater than 0.8.

Finally, we carried out a comparison of the existing *SO* middleware solutions available. We focused the comparison in the support these solutions give to the set of fundamental processes along the key areas defined in the *E-Ma-Gen3* analysis tool. We observed that *em4so* middleware offers support to the mentioned processes without constraining the autonomy relative to the user, platforms or other *SOs*. This is a key difference of *em4so* in regards compared solutions.

Chapter 8

Conclusion and Future Work

In summary this thesis has presented a framework for development of smart objects based on the concept of relative autonomy. Our overall research challenges are (1) the examination of the smart object's properties and definition of autonomy under this context, (2) the definition of a software architecture, for *SOs*, based on the mentioned concepts, (3) development of a middleware architecture that gathers common functionalities of the software architecture, (4) Definition of a method for adaptation of the smart objects at individual and collective level; and (5) an approach for the evaluation of smart objects systems under different situations of heterogeneity, volatility and large quantities of nodes.

In this chapter we will discuss our research contributions in addressing the mentioned challenges. We also present some potential future directions to extend further the present work.

8.1 Research Contributions

8.1.1 Foundations of smart object autonomy

We examined the smart object characteristics from five key areas —knowledge, behaviour, relationships, resource and structure—, three planes —cyber, physical and social— and three fundamental processes —exploitation, management and generation— . We concluded that autonomy implies that the smart object is able to carry out its fundamental processes on the elements and planes identified, in an independent way. The autonomy of the *SO* is constrained by structural, resource, knowledge and cooperating dependencies. While the former dependency is desirable in cooperative goal achieve-

ment scenarios, the other three former dependencies must be avoided for development of the potential autonomy. We also defined collective of smart objects as a system and a society of heterogeneous smart objects that cooperate towards common goals.

8.1.2 A software architecture for smart objects

Based on the concept of smart object's autonomy we defined a software architecture or building *IoT* applications: *Role-based Smart Objects* (*RbSOs*) . The building blocks of *IoT* applications are goal-oriented and role-based smart objects. Key abstractions of this architecture are organised in terms of an uncoupled goal-oriented behaviour and a knowledge representation. Individual *SO*'s capabilities are wrapped as services and used as actions —with concrete execution arguments— of a set of plan activities. The roles are defined in terms of these activities and work as high level interface of the *SO* behaviour, that enables other *SOs* to locate potential cooperating partners. We demonstrated the feasibility of our approach by the implementation of case study for provisioning of physical resources.

8.1.3 A embedded middleware architecture for smart objects

From the abstractions defined previously we proposed a middleware architecture (*em4so*) that incorporates the components covering common functionalities in smart object application development. The *em4so* architecture is distributed in Governing and Management bodies. The former includes the components for decision-making by the *SO*, namely Smart Object Controller, Knowledge Base and Reasoning Engine. The later is formed by the Capability Manager, KB and Storage Manager and the Social Interaction Manager. Bodies are extended by a set of communication, device and extra facilities. The architecture also defines a p2p smart object protocol that enables creation of *SO* overlay networks, queries, cooperation and coordination for achieving common goals. We implemented a middleware prototype using the architecture and adopt it for evaluation, by the development of the case study and a simulation. We compared our middleware architecture with that of other existing solutions, we concluded that a key difference is the focus and support we offer to autonomy not only related to human user, but also to platform and other *SOs*. We demonstrated the scalability of this architecture facing increasing number of *SOs* and increasing number of concurrent plans.

8.1.4 A method for adaptation of smart objects systems at individual and collective level

We presented our strategy for adaptation based in the dynamic selection of *SOs*, services and roles according to the context and the user preferences. For the selection, we defined a set of utility functions that enable to assess different candidates and so identify the best suited for a given context. Every time the context changes, these utilities are recalculated enabling to always obtain the best available candidates, namely *SOs*, services or roles. These utilities are calculated considering the ranked preferences of human users in regards four key group of factors: trustworthiness, performance, efficiency and location. We evaluated the adaptation method at low scale using the case study defined and a higher scale using a simulated environment based on an agent-based model. We observed how this method enables the *SO* systems to adapt and show resilience facing unavailability of some of the nodes. Likewise, the method enables that new *SOs* coming to the collective of *SOs* are considered for carry out cooperative goals.

8.1.5 An agent-based model for evaluation of smart object systems

We developed an agent-based model that mimics the operation of a collective of smart objects. *SOs* are modelled as heterogeneous agents that have different hardware configurations and capabilities. On top of each *SO*, software might be installed that enables the *SOs* to carry out goals, communicate and coordinate between each other. For our evaluation, we installed in each *SO* the *em4so* middleware (and *SO* protocol). We defined two metrics that enabled measuring performance of *SO* collective at different situations, namely: *Mean Query Time* and *Plan Success Rate*. We showed the application of the model in the evaluation of scalability and adaptability of the *em4so* middleware architecture.

From the whole evaluation, we conclude that the development of *RbSOs* using *em4so* architecture is a feasible alternative to existing *IoT* development approaches.

8.2 Future Work

We identify here some of the potential work streams that can be taken in order to continue our work.

8.2.1 Machine Learning Services to *SO* Middleware

We have learned that one approach to achieve generation of knowledge is through the use of machine learning techniques including supervised, unsupervised and deep learning. Our current middleware architecture does not include by default use of machine learning techniques for decision making, however these can be included as extra facilities services and made available for different *SO* applications. These can be used concretely to extend the Reasoning Engine component of the architecture (section 5.5.3) and to offline classify the different *QoS* attributes or usage patterns that need to be considered in a group criteria, according to the service selection method described in section 6.6.

8.2.2 *RbSOs* through Unikernels

We have defined the abstractions for *RbSOs* software and made concrete, through functional components of a middleware architecture, those that are common to multiple *IoT* applications. Another approach to make these abstractions concrete would be to incorporate the functional components as part of the operating system libraries of a Unikernel system. Unikernels have become popular in the past years, these run on top of an hypervisor, E.g. XEN [1] and enable to have a specific customised kernel for the application running on top of it, instead of a general purpose operating system [77]. Unikernel only contains the drivers and libraries required for the application, becoming more secure as reduce the potential threads derived from, by default, services available in a modern operating system but not used for the particular application E.g. support to a particular communication protocol. Therefore, a kind of agent-based unikernel could be used to implement a single bootable virtual machine containing the *Smart Object Controller* and the rest of the architecture components. It is open to investigate how the middleware architecture components could be transformed into operating system libraries.

8.2.3 Real-time *RbSOs* Application Development

One key potential application of *RbSOs* is for smart healthcare and smart factory systems. These systems have strict privacy and confidentially requirements, respectively, that can be addressed following the proposed approach. However, these system have also tough real-time processing requirements, in this domain, any delay might cause

[1] https://www.xenproject.org/

life costs, so these are not tolerated. Since in our real-setting evaluation, our prototype experienced delays mainly by from sensor readings, as it is, is not fit for these kind of systems. It is open to explore and evaluate the optimisation required at implementation level to make this fit these kind of systems and to determine if there is any improvement to be made at architectural level. This would require selection of efficient programming language and platforms for implementation.

8.2.4 Hierarchical P2P *SO* Protocol

Our approach for the cooperation and coordination *SO* protocol is based in a simple p2p unstructured overlay. In hierarchical overlays, the nodes are organised following parent/child relationships. This would allow to organise all the *SOs* able to play a role around a set of powerful *SOs* as super peers. The super peers would canalise query requests for a particular role and store information about role density among the overlay. This information would allow to extend the partial role density measurement (section 6.7), providing more accurate information for role selection. These super peers have to define mechanisms of replication in order to avoid becoming a source of dependency to the other *SOs* of the collective.

8.2.5 Blockchain for Activity Tracking

Blockchain enables the storage of transactions in a ledger that is managed by an open decentralised p2p infrastructure. Transactions regarding the activities that every *SO* carries out might be stored in this ledger enabling the access to the authorised *SOs* with purpose of analysis and tracking of *SO* operation. In addition, blockchain extra facilities could be implemented to enable backup of the transformed data gathered by each *SO*.

Appendices

Appendix A

em4so's Middleware Prototype Implementation Examples

A.1 Service List by Host: Map Function

```
function (doc) {
  var argValues = [];
  var hostKeys = {};
  var result = null;
  var k;
  if (doc.type && doc.type == 'service'){
    for (var j in  doc.args){
      argValues.push(doc.args[j]);
    }
    if(!doc.result || doc.result==null || doc.result == "")
      result = "";
    else
      result = doc.result;
    hostKeys = Object.keys(doc.host);
    for(var i=0,l=hostKeys.length;i<l;i++){
      k = hostKeys[i];
      emit([doc.name,result,argValues], {id:k, capability: doc.host[k
          ].capability, ranking:doc.host[k].ranking},1);
    }
  }
}
```

A.2 Service List by Host: Reduce Function

```
function (keys, values,rereduce) {
  if (values.length > 1) {
    var max = 0, ks = values;
    for (var i = 1, len = ks.length; i < len; ++i) {
      if (ks[max].ranking < ks[i].ranking) {
        max = i;
      }
    }
    return ks[max];
  } else {
    return values[0];
  }
}
```

A.3 Preconceived belief: Device

```
{
   "_id": "device/parameters",
   "type": "parameters",
   "parameters": [
       {
           "type": [
               "level",
               "temperature",
               "air_quality"
           ]
       },
       {
           "mode": [
               "active",
               "passive"
           ]
       }
   ]
}
```

A.4 Preconceived belief: High

```
{
"type":"concept",
"_id":"concept/high",
"name": "high",
"equivalent":[
"100","TRUE"
] ,
"leadedby":"increase",
"makesvalid":["greatherthan","greatherequalthan"
]
}
```

Appendix B

Case Study Implementation Examples

B.1 JSON Activity Definition

```
{ "_id": "activity/keepAirFresh",
  "type": "activity",
  "categories": [..],
  "input": {"knowledge":[{
        "scope": "saf_agent",
        "name": "freshener",
        "kind":"resource",
        "attrNames":
        [ "model" ] }],
    "operator": "AND",
    "operands": [{"operator": ">=",
            "operand1": {"scope": "saf_agent",
          "name": "freshener",
          "kind":"resource",
          "attributeName": "level"},
            "operand2": { "value": "10" }},
           {"operator": "==",
            "operand1": {"scope": "room",
          "name": "activity",
          "kind":"feature",
          "attributeName":"level"},
            "operand2": {"value": "true"}}]},
  "description": "actions related to keep the air fresh "
     ,
  "actions":  [{  "name": "spray",
          "args": []}],
  "output":[{   "scope": "saf_agent",
          "name": "freshener",
          "kind": "resource",
          "effect": "decrease"},
        { "scope": "room",
          "name": "freshness",
          "kind": "feature",
          "effect": "increase",
          "attrNames":["level"] }],
  "quality": {}}
```

B.2 JSON Scenario Definition

```
{
   "_id": "scenario/keepHomeResources",
   "type": "scenario",
   "goal": "goals/keepHomeResources",
   "description":"scenario for achieving goal keep home
      resources",
   "steps":[
   {"activity":"activity/increaseHomeDemand"},
   {"activity":"activity/askForRepurchase"},
   {"activity":"activity/askNotifyUser"}
   ]

}
```

Bibliography

[1] Mohammad Aazam and Eui Nam Huh. Fog Computing: The Cloud-IoT/IoE Middleware Paradigm. *IEEE Potentials*, 35(3):40–44, 2016.

[2] Gregory D Abowd and Elizabeth D Mynatt. Charting past, present, and future research in ubiquitous computing. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):29–58, 2000.

[3] Francesco Aiello, Giancarlo Fortino, Raffaele Gravina, and Antonio Guerrieri. A java-based agent platform for programming wireless sensor networks. *Computer Journal*, 54(3):439–454, 2011.

[4] G B Al-Suwaidi and M J Zemerly. Locating friends and family using mobile phones with global positioning system (GPS). *2009 IEEE/ACS International Conference on Computer Systems and Applications*, pages 555–558, 2009.

[5] Mohammad Alrifai, Dimitrios Skoutas, and Thomas Risse. Selecting skyline services for QoS-based web service composition. *Proceedings of the 19th international conference on World wide web*, 2588(5):11–20, 2010.

[6] W Alshabi, S Ramaswamy, and M Itmi. Coordination, cooperation and conflict resolution in multi-agent systems. *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, pages 495–500, 2007.

[7] Qazi Mamoon Ashraf, Mohamed Hadi Habaebi, and Md. Rafiqul Islam. TOPSIS-Based Service Arbitration for Autonomic Internet of Things. *IEEE Access*, 4:1313–1320, 2016.

[8] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, oct 2010.

[9] Luigi Atzori, Antonio Iera, and Giacomo Morabito. SIoT: Giving a social structure to the internet of things. *IEEE Communications Letters*, 15(11):1193–1195, 2011.

[10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. Understanding the Internet of Things: definition, potentials, and societal role of a fast evolving paradigm. *Ad Hoc Networks*, 56:122–140, 2016.

[11] Luigi Atzori, Antonio Iera, Giacomo Morabito, and Michele Nitti. The social internet of things (SIoT) - When social networks meet the internet of things: Concept, architecture and network characterization. *Computer Networks*, 56(16):3594–3608, 2012.

[12] Panos Bardis. Social Interaction and Social Processes. *Social Science*, 54(3):147–167, 1979.

[13] Martin Bauer, Mathieu Boussard, Nicola Bui, and Francois Carrez. Project Deliverable D1.2 âĂŞ Final Architectural Reference Model for IoT. Technical report, IoT-A Project - UniS, jul 2013.

[14] Tobias Betz, Lawrence Cabac, and M Wester-Ebbinghaus. Gateway architecture for Web-based agent services. *Multiagent System Technologies*, pages 165–172, 2011.

[15] Giulia Biamino. A Semantic Model for Socially Aware Objects. *Advances in Internet of Things*, 02(03):47–55, 2012.

[16] Julian Bleecker. A Manifesto for Networked Objects âĂŤ Cohabiting with Pigeons , Arphids and Aibos in the Internet of Things. Technical report, University of Southern California, 2005.

[17] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[18] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*, volume 1. Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[19] Giacomo Cabri, Letizia Leonardi, Luca Ferrari, and Franco Zambonelli. Role-based software agent interaction models: a survey. *The Knowledge Engineering Review*, 25(04):397–419, 2010.

[20] Cosmin Carabelea and Olivier Boissier. Multi-agent platforms on smart devices: Dream or reality. In *Proceedings of the Smart Objects Conference (SOC03), Grenoble, France*, pages 126–129. Citeseer, 2003.

[21] Cristiano Castelfranchi and Rino Falcone. Founding autonomy: The dialectics between (social) environment and agent's architecture and powers. In *Agents and Computational Autonomy*, pages 40–54. Springer, 2003.

[22] Cristiano Castelfranchi and Rino Falcone. From Automaticity to Autonomy: The Frontier of Artificial Agents. *Agent Autonomy*, pages 103–136, 2003.

[23] Humberto Cervantes and R.S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. *Proceedings. 26th International Conference on Software Engineering*, 3:2–11, 2004.

[24] Eliseo Clementini, Paolino Felice, and Peter Oosterom. A small set of formal topological relationships suitable for end-user interaction. *Advances in Spatial Databases*, pages 277–295, 1993.

[25] Alan Colman and Jun Han. Roles, players and adaptable organizations. *Applied Ontology*, 2(Number 2/2007):105–126, 2007.

[26] Mark S. Daskin. *Service Science : Service Operations for Managers and Engineers*. John Wiley & Sons, Ltd, Hoboken, N.J, 2011.

[27] Rogério De Lemos, Holger Giese, and et Al. Software engineering for self-adaptive systems: A second research roadmap. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7475 LNCS:1–32, 2013.

[28] Simon Duquennoy, Gilles Grimaud, and Jean Jacques Vandewalle. The web of things: Interconnecting devices with high usability and performance. *Proceedings - 2009 International Conference on Embedded Software and Systems, ICESS 2009*, pages 323–330, 2009.

[29] Emilio Ferrara, Onur Varol, Clayton Davis, Filippo Menczer, and Alessandro Flammini. The Rise of Social Bots. *arXiv preprint arXiv:1407.5225*, 59(7):1–11, 2016.

[30] FIPA. Fipa Abstract Architecture Specification. Technical report, Foundation for Intelligent Physical Agents, 2002.

[31] FIPA. FIPA Agent Management Specification. Technical report, FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS, 2004.

[32] Giancarlo Fortino, A. Guerrieri, and W. Russo. Agent-oriented smart objects development. In *Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on*, pages 907–912, 2012.

[33] Giancarlo Fortino, Antonio Guerrieri, and Michelangelo Lacopo. An agent-based middleware for cooperating smart objects. *Highlights on Practical ...*, pages 387–398, 2013.

[34] Giancarlo Fortino, Antonio Guerrieri, Wilma Russo, and Claudio Savaglio. Middlewares for smart objects and smart environments: overview and comparison. In *Internet of Things Based on Smart Objects*, pages 1–27. Springer, 2014.

[35] Giancarlo Fortino, Marco Lackovic, Wilma Russo, and Paolo Trunfio. A discovery service for smart objects over an agent-based middleware. In *Internet and Distributed Computing Systems*, pages 281–293. Springer, 2013.

[36] Giancarlo Fortino, Wilma Russo, and Claudio Savaglio. Agent-oriented Modeling and Simulation of IoT Networks. In *Federated Conference on Computer Science and Information Systems*, volume 8, pages 1449–1452, 2016.

[37] Giancarlo Fortino and Paolo Trunfio. Internet of Things Based on Smart Objects. *Springer*, 2014.

[38] Justin Frankel and T. Pepper. The Gnutella Protocol Specification v0.4 1. Technical report, Clip2, 2003.

[39] Emilia Garcia, Adriana Giret, and Vicente Botti. Software engineering for service-oriented MAS. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5180 LNAI:86–100, 2008.

[40] Michael P. Georgeff and Francois Felix Ingrand. Decision-Making in an Embedded Reasoning System. *Proceedings of the 11th international joint conference on Artificial intelligence*, 2:972–978, 1989.

[41] Christos Goumopoulos and Achilles Kameas. Smart objects as components of UbiComp applications. *International Journal of Multimedia and Ubiquitous Engineering*, 4(3):1–20, 2009.

[42] Dominic Greenwood, Margaret Lyell, Ashok Mallya, and Hiroki Suguri. The IEEE FIPA approach to integrating software agents and web services. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems - AAMAS '07*, volume 5, page 1, 2007.

[43] Dominique D. Guinard and M.Trifa Vlad. Building the Web of Things. *Manning Publications*, 2, 2015.

[44] Cesar Gutierrez, Juan Garbajosa, Jessica Diaz, and Agustin Yague. Providing a Consensus Definition for the Term" Smart Product". In *Engineering of Computer Based Systems (ECBS)*, pages 203–211. IEEE, apr 2013.

[45] J. Octavio Gutierrez-Garcia and Kwang Mong Sim. Agent-based cloud service composition. *Applied Intelligence*, 38(3):436–464, 2013.

[46] Serge Haddad, Amal El, and Fallah Seghrouchni. Web-MASI : Multi-Agent Systems Interoperability. In *2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology(IAT'05)*, pages 2–5, 2005.

[47] Henry Hexmoor. A model of absolute autonomy and power: Toward group effects. *Connection Science*, 14(4):323–333, 2002.

[48] Henry Hexmoor, Cristiano Castelfranchi, and Rino Falcone. *Agent Autonomy*, volume 1. Springer Science + Business Media, LLC, 2003.

[49] Lars Erik Holmquist, Friedemann Mattern, Bernt Schiele, Petteri Alahuhta, Michael Beigl, and Hans-W Gellersen. Smart-its friends: A technique for users to easily establish connections between smart artefacts. In *Ubicomp 2001: Ubiquitous Computing*, pages 116–122. Springer, 2001.

[50] Ivar Jacobson, Grady Booch, and James Rumbaugh. The Unified Software Development Process. *IEEE Software*, 16:96–102, 1999.

[51] JADE. JADE Web Service Dynamic Client Guide. Technical report, CSELT S.p.A, TILab S.p.A., 2010.

[52] JADE. JADE Web Service Integration Gateway (WSIG). Technical report, Telecom Italia, 2015.

[53] Anil K. Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651–666, 2010.

[54] Jean-paul Jamont, M Lionel, and Michael Mrissa. A Web-Based Agent-Oriented Approach to Address Heterogeneity in Cooperative Embedded Systems. *Advances in Intelligent Systems and Computing. Trends in Practical Applications of Heterogeneous Multi-Agent Systems. The PAAMS Collection*, pages 45–52, 2014.

[55] Xiongnan Jin, Sejin Chun, Jooik Jung, and Kyong Ho Lee. IoT service selection based on physical service model and absolute dominance relationship. *Proceedings - IEEE 7th International Conference on Service-Oriented Computing and Applications, SOCA 2014*, pages 65–72, 2014.

[56] Jos De Jong, Ludo Stellingwerff, and Giovanni E. Pazienza. Eve: A Novel Open-Source Web-Based Agent Platform. In *2013 IEEE International Conference on Systems, Man, and Cybernetics*, pages 1537–1541. Ieee, oct 2013.

[57] Euihyun Jung, Ilkwon Cho, and Sun Moo Kang. iotSilo: The Agent Service Platform Supporting Dynamic Behavior Assembly for Resolving the Heterogeneity of IoT. *International Journal of Distributed Sensor Networks*, 2014:1–11, 2014.

[58] Stephen H. Kaisler. *Software paradigms*. John Wiley \& Sons, 2005.

[59] Paul Karaenke, Michael Schuele, András Micsik, and Alexander Kipp. Inter-organizational interoperability through integration of multiagent, web service, and semantic web technologies. *Lecture Notes in Business Information Processing*, 98 LNBIP:55–75, 2012.

[60] Artem Katasonov, Olena Kaykova, Oleksiy Khriyenko, Sergiy Nikitin, and Vagan Terziyan. Smart Semantic Middleware for the Internet of Things. *ICINCO-ICSO*, pages 169–178, 2008.

[61] Artem Katasonov and Vagan Terziyan. SmartResource Platform and Semantic Agent Programming Language (S-APL). *Multiagent System Technologies*, pages 25–36, 2007.

[62] Fahim Kawsar. A Document-Based Framework for User Centric Smart Object Systems. *PhD in Computer Science, Waseda University, Japan*, 0:140, 2009.

[63] Fahim Kawsar and Tatsuo Nakajima. A document centric framework for building distributed smart object systems. *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2009*, pages 71–79, 2009.

[64] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[65] JO Kephart and DM Chess. The vision of autonomic computing. *Computer*, 36(January):41–50, 2003.

[66] El Mehdi Khalfi, Jean Paul Jamont, Michael Mrissa, and Lionel Medini. A RESTful task allocation mechanism for the Web of Things. *2016 IEEE RIVF International Conference on Computing and Communication Technologies: Research, Innovation, and Vision for the Future, RIVF 2016 - Proceedings*, pages 73–78, 2016.

[67] Mohamed Essaid Khanouche, Yacine Amirat, Abdelghani Chibani, Moussa Kerkar, and Ali Yachir. Energy-Centered and QoS-Aware Services Selection for Internet of Things. *IEEE Transactions on Automation Science and Engineering*, 13(3):1256–1269, 2016.

[68] Ji Eun Kim, Adriano Maron, and Daniel Mosse. Socialite: A Flexible Framework for Social Internet of Things. *2015 16th IEEE International Conference on Mobile Data Management*, pages 94–103, 2015.

[69] Gerd Kortuem, Fahim Kawsar, Daniel Fitton, and Vasughi Sundramoorthy. Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1):44–51, 2010.

[70] Bent Bruun Kristensen and Kasper Østerbye. Roles: conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.

[71] Philippe Lalanda, Julie A. McCann, and Ada Diaconescu. *Autonomic Computing Principles, Design and Implementation*. Springer, London, UK, 1 edition, 2014.

[72] Edward A. Lee. Computing Foundations and Practice for Cyber- Physical Systems : A Preliminary Report. *Electrical Engineering and Computer Sciences University of California at Berkeley*, pages 1–27, 2007.

[73] Teemu Leppänen and Jukka Riekki. A lightweight agent-based architecture for the Internet of Things. In *IEICE workshop on Smart Sensing, Wireless Communications, and Human Probes*, pages 2–4, 2013.

[74] Teemu Leppänen, Jukka Riekki, Meirong Liu, Erkki Harjula, and Timo Ojala. Internet of Things Based on Smart Objects. In Giancarlo Fortino, editor, *Internet of Things Based on Smart Objects*, pages 29–48. Springer International Publishing, 2014.

[75] Michael Luck, Mark D'inverno, and Steve Munroe. Autonomy: Variable and Generative. In Henry Hexmoor, Cristiano Castelfranchi, and Rino Falcone, editors, *Agent Autonomy*, chapter 2, pages 11–28. Kluwer Academic Publishers, 2003.

[76] Jianhua Ma. Smart u-Things - Challenging Real World Complexity. In *IPSJ Symposium Series*, volume 19, pages 146–150, Japan, nov 2005.

[77] Anil Madhavapeddy and David J. Scott. Unikernels: The Rise of the Virtual Library Operating System. *ACM Queue - Distributed Computing*, 11(11):30, 2013.

[78] Friedemann Mattern. From smart devices to smart everyday objects. In *Proceedings of smart objects conference*, 2003.

[79] Lionel Médini, Michael Mrissa, El-Mehdi Khalfi, Mehdi Terdjimi, Nicolas Le Sommer, Philippe Capdepuy, Jean-Paul Jamont, Michel Occello, and Lionel Touseau. *Chapter 5 - Building a Web of Things with Avatars: A comprehensive approach for concern management in WoT applications*. Elsevier Inc., 1 edition, 2017.

[80] Roberto Minerva, Abyi Biru, and Domenico Rotondi. Towards a definition of the Internet of Things (IoT). *IEEE Internet Things*, pages 1–86, 2015.

[81] Mark Moriconi and Robert Riemenschneider. Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Technical report, SRI International - Computer Science Laboratory, 1997.

[82] Salama A. Mostafa, Mohd Sharifuddin Ahmad, and Aida Mustapha. Adjustable autonomy: a systematic literature review. *Artificial Intelligence Review*, pages 1–38, 2017.

[83] Michael Mrissa and Nicolas Le Sommer. An Avatar Architecture for the Web of Things. *IEEE Internet Computing*, 2015.

[84] Max Mühlhäuser. Smart Products : An Introduction. In *Constructing Ambient Intelligence Workshops*, pages 158–164. Springer, 2008.

[85] Computer Networks. SmartSantander: IoT Experimentation over a Smart City Testbed. *Computer Networks*, 61(November):217–238, 2015.

[86] OGC. SensorThings Data Model, 2013.

[87] Elizabeth Papadopoulou, Sarah Gallacher, Nick K. Taylor, and M. Howard Williams. Personal smart spaces as a basis for identifying users in pervasive systems. *Proceedings - Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing in Conjunction with the UIC 2010 and ATC 2010 Conferences, UIC-ATC 2010*, pages 88–93, 2010.

[88] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[89] Charith Perera, A. Zaslavsky, P. Christen, M. Compton, and D. Georgakopoulos. Context-Aware Sensor Search, Selection and Ranking Model for Internet of Things Middleware. In *IEEE International Conference on Mobile Data Management*, volume 1, pages 314–322, 2013.

[90] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context Aware Computing for The Internet of Things: A Survey. *Communications Surveys & Tutorials*, 2013.

[91] Marco E Perez Hernandez and Stephan Reiff-Marganiec. Classifying smart objects using capabilities. In *Proceedings of 2014 International Conference on Smart Computing, SMARTCOMP 2014*, pages 309–316, 2014.

[92] Marco E Pérez Hernández and Stephan Reiff-Marganiec. Autonomous and self-controlling smart objects for the future internet. In *2015 3rd International Conference on Future Internet of Things and Cloud (FiCloud)*, 2015.

[93] Marco E Perez Hernandez and Stephan Reiff-marganiec. Towards a software framework for the autonomous internet of things. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2016.

[94] Zheng Qin, Xiang Zheng, and Jiankuan Xing. *Software Architecture*. Springer, 2008.

[95] Stephan Reiff-Marganiec, Hong Qing Yu, and Marcel Tilly. Service selection based on non-functional properties. In *Service-Oriented Computing-ICSOC 2007 Workshops*, pages 128–138. Springer, 2009.

[96] Debbie Richards, Sander van Splunter, Frances M T Brazier, and Marta Sabou. Composing Web services using an agent factory. *1st Workshop on Web Services and Agent-Based Engineering, WSABE'03*, pages 57–66, 2003.

[97] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly, Sebastopol, CA, first edition, 2008.

[98] Luis Roalter, Matthias Kranz, Andreas Möller, and Technische Universität München. A middleware for intelligent environments and the internet of things. *in Ubiquitous Intelligence and Computing*, pages 267–281, 2010.

[99] Albrecht Schmidt, Michael Beigl, and Hans-W Gellersen. There is more to context than location. *Computers & Graphics*, 23(6):893–901, dec 1999.

[100] Quan Z. Sheng, Xiaoqiang Qiao, Athanasios V. Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218–238, 2014.

[101] Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. Role Of Middleware For Internet Of Things: A Study. *International Journal of Computer Science & Engineering Survey*, 2(3):94–105, aug 2011.

[102] Friedrich Steimann. The Role Data Model Revisited. *Applied Ontology*, 2(Bachman 1973):89–103, 2005.

[103] Bruce Sterling. *Shaping Things*, volume 39. MIT Press, Cambridge, Mass., 2005.

[104] Leon Sterling and Kuldar Taveter. *The Art of Agent-Oriented Modeling*. The MIT Press, feb 2009.

[105] Harald Sundmaeker, Patrick Guillemin, Peter Friess, and S Woelfflé. *Vision and challenges for realising the Internet of Things*. EUR-OP, Luxembourg, 2010.

[106] Mehdi Terdjimi, Lionel Médini, Michael Mrissa, and Maria Maleshkova. Multi-purpose Adaptation in the Web of Things. In *International and Interdisciplinary Conference on Modeling and Using Context*, volume 3554, pages 213–226. Springer Cham, 2017.

[107] Mehdi Terdjimi, Lionel Medini, Michael Mrissa, and Nicolas Le Sommer. An avatar-based adaptation workflow for the web of things. *Proceedings - 25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2016*, pages 62–67, 2016.

[108] H. Tschofenig, J. Arkko, D. Thaler, and D. McPherson. Architectural Considerations in Smart Object Networking. Technical report, Internet Architecture Board, 2015.

[109] Dieter Uckelmann, Mark Harrison, and Florian Michahelles. *Architecting the Internet of Things*. Springer, 2011.

[110] Luis M. Vaquero and Luis Rodero-Merino. Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[111] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting Smart Objects with IP: The Next Internet - Interconnecting Smart Objects with IP.pdf*. Elsevier-Morgan Kaufmann, Burlington, MA, 2010.

[112] Ovidiu Vermesan and Peter Friess. *Building the Hyperconnected Society*. River Publishers, 2015.

[113] David. Vernon. *Artificial Cognitive Systems: A Primer*. The MIT Press, 2014.

[114] Mitchell Waldrop. More Than Moore. *Nature*, 530(7589):145, 2016.

[115] Michael E Whitman and Herbert J Mattord. *Principles of Information Security*. Cengage Learning, Boston, MA, fourth edition, 2011.

[116] C.Y Chien Yaw Wong, Duncan McFarlane, D. Zaharudin, V. Awarwal, A Ahmad Zaharudin, and Vivek Agarwal. The Intelligent Product Driven Supply Chain. In *IEEE International Conference on systems, man and cybernetics*, volume 4, page 6. IEEE, 2002.

[117] Michael Wooldridge. *Introduction to Multiagent Systems*, volume 30. Wiley & Sons, Ltd, Glasgow, second edition, 2009.

[118] M Wooldridgey and Paolo Ciancarini. Agent-oriented software engineering: The state of the art. *Agent-Oriented Software Engineering*, 2001.

[119] Qihui Wu, Guoru Ding, Yuhua Xu, Shuo Feng, Zhiyong Du, Jinlong Wang, and Keping Long. Cognitive Internet of Things: A New Paradigm Beyond Connection. *IEEE Internet of Things Journal*, 1(2):129–143, 2014.

[120] Zhaohui Wu, Shuiguang Deng, and Jian Wu. Service-Oriented Architecture and Web Services. In *Service Computing Concepts, Methods and Technology*, volume 9, chapter 2, pages 62–64. Elsevier Inc., 2015.

[121] Haiping Xu. Developing Role-Based Open Multi-Agent Software Systems. *International Journal of Computational Intelligence Theory and Practice*, 2(1), 2007.

[122] Hong Qing Yu and Stephan Rei. Non-functional property based service selection: A survey and classi cation of approaches. In *Service Oriented Computing Workshop. The 6th IEEE European Conference on Web Services*, 2008.

[123] John a Zachman. A framework for information systems architecture. *IBM Systems Journal*, 38(2):454, 1999.

[124] John A Zachman. The Framework for Enterprise Architecture: Background, Description and Utility by: John A. Zachman. *Retrieved from: https://www.zachman.com/resources/ea-articles-reference/327-the-framework-for-enterprise-architecture-background-description-and-utility-by-john-a-zachman*, pages 1–5, 2016.

[125] Deze Zeng, Song Guo, and Zixue Cheng. The Web of Things: A Survey (Invited Paper). *Journal of Communications*, 6(6), sep 2011.

[126] Haibin Zhu. Role-Based Collaboration and E-CARGO. *IEEE Systems, Man & Cybernetics*, pages 27–35, jul 2015.

[127] Ingo Zinnikus, Gorka Benguria, Brian Elvesaeter, Klaus Fischer, and Julien Vayssiere. A Model Drive Approach to Agent-Based Service-Oriented Architectures. In *Multiagent System Technologies*, volume LNAI 4196, pages 110–122. Springer, 2006.