



**ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA**

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

**From data to applications
in the Internet of Things**

Tesi in Ingegneria dei Sistemi Software

Candidato:

Ing. Simone Norcini

Matricola 0000652524

Relatore:

Ch.mo Prof. Ing. Antonio Natali

From data to applications in the Internet of Things

Ing. Simone Norcini

July 6, 2016

Contents

1	Introduction	1
2	Cloud Computing	5
2.1	Origin of the Cloud	5
2.1.1	SOAP vs REST: final considerations	8
2.2	Infrastructure as a Service (IaaS)	9
2.3	Platform as a Service (PaaS)	10
2.4	Software as a Service (SaaS)	11
2.5	Cloud types	11
2.5.1	Public Cloud	11
2.5.2	Private Cloud	12
2.5.3	Hybrid Cloud	13
3	Cloud service providers and software development	15
3.1	Virtualization	15
3.1.1	Openstack	16
3.1.2	Cloud Foundry	17
3.2	DevOps	19
3.2.1	DevOps best practices	20
3.2.2	DevOps system components	21
3.3	Microservice Architecture	23
3.4	Code portability between providers	24
3.4.1	Unportable PaaS	25
3.4.2	Portable PaaS	26
3.5	Providers comparison	30

4 Fog Computing	35
4.1 Fog systems features	36
5 Internet of Things	39
5.1 IoT definitions	40
5.2 Ubiquitous Computing	41
5.3 Constituent elements of the IoT	42
5.4 IoT focused on the Cloud	43
5.5 IoT focused on the Fog	44
5.6 MQTT	46
5.6.1 MQTT features	46
5.6.2 MQTT methods	47
6 Case Study	49
6.1 Product requirement specification	
STEP 1	49
6.1.1 Business requirements	49
6.1.2 Architectural requirements	49
6.1.3 Scenario	50
6.1.4 Functional requirements	50
6.1.5 Non functional requirements	51
6.2 Product requirement specification	
STEP 2	51
6.3 Analysis - STEP 1	51
6.3.1 QActors	53
6.3.2 Workflow for QActors - Phase 1	56
6.3.3 Workflow for QActors - Phase 2	56
6.3.4 Workflow for QActors - Phase 3	56
6.3.5 Workflow for QActors - Phase 4	57
6.3.6 Workflow for QActors - Phase 5	58
6.3.7 Workflow for QActors - Phase 6	61
6.4 The system running	68
7 Conclusions	69

CONTENTS

v

Bibliography

71

Chapter 1

Introduction

With the growth in complexity of the IT infrastructure, which every day becomes more and more pervasive, there is a need of new computational models capable of supporting scenarios never faced before. The pervasiveness and the geolocation are implicit in an *Internet of things* [18] scenario leading to the need to rethink the placement of information within the system components. The rapid growth of computational capacity of *System on a Chip* (SoC) also suggests a redeployment of system's informations downward and the emergence of a multiplicity of independent entities which can acquire the knowledge necessary to accomplish their goals and make decisions that can lead to a change of state of the world.

To support lots of heterogeneous and distributed entities comes the need of new paradigms that may offer a geolocation support and data management services near to the data source aiming at reducing the data bandwidth demand as the *Fog Computing* [25] promises.

The exponential information increase generated by ubiquitous devices with more and more sensors also leads to the need to rethink policies for the data centers management and the need to support complex analytics services the output of which may have to be viewed by millions of users at a time. For this purpose *Cloud Computing* [13] solutions have recently gained more and more attention, and several

vendors are considering with interest the feasible solutions in order to optimize the use of their hardware infrastructure. In fact, among the different benefits, these solutions provide the computational resources as virtualized entity allowing to dramatically reduce the initial capital investment and maintenance costs. So, the main force of the cloud is to offer computational resources to third-party service providers who have no intention of building and maintaining their own IT infrastructure.

It is a paradigm shift [7] understood as a change in the fundamental chain of events. After the transition from mainframe to client-server architecture, the next hop it leads to the *Cloud*. Thanks to this new paradigm, users can abstract the technological details that relate to the infrastructure hardware and no longer need the skills or control over the technology infrastructure as the underlying cloud supports them.

This new scenario leads us to rethink the way in which the software is designed and developed in an agile perspective. Cloud based software development by Dev teams should be closely related to the activities of the Ops team that supports the Cloud creating a new philosophy called *DevOps* [12]. Trying to achieve DevOps in an organization we should address the structural implications because the past architectures may be inadequate or incomplete to bridge the abstraction gap. DevOps has implications with respect to both the overall structure of the system and techniques that should be used in the system's element [19].

With the lack of an adequate abstraction supported at language level, developers in the IoT field remain with the only possibility of making a bottom-up construction with the aggravating factor of extreme heterogeneity. However, in order to help the developers, a new architectural style is advocated first. The works are in progress trying to formalize what *microservices* are [22] because, as a general perspective, the monolithic applications of the past appear to be hardly scalable and maintainable in a Cloud environment.

To manage and organize every model which arises from the IoT wide scenario into a single framework, the chance to think about models as a *new form of source code* would be invaluable. Also having generators able to make executable code automatically from framework's models would be useful. In this case, modifying the models would be enough to change the framework [21]. *Model Driven Software Development* (MDS) approach aims to reduce transformations complexity restricting changes to the "schematic" and repetitive part of a specific platform. The starting point of this approach consists in defining a *Domain Specific Language* (DSL) by the use of formal models that are processed directly in the code or in the other artifacts provided by the platform. The IoT experiment accomplished has been designed in such a way to take *meta concepts* emerging from the Internet of Things environment and injecting them in a *Model Driven software factory* in order to get an automatic code generation and regaining the top-down approach.

In this thesis we will start by analyzing the *Cloud Computing* model in chapter 2 and the concrete implications of this model on the world's providers in chapter 3. As a possible alternative to the cloud, or as intermediate structure capable of providing support to highly geolocated systems, it has been recognized the paradigm of *Fog Computing* in chapter 4. To conclude the theoretical part and to introduce the project was finally introduced in chapter 5 the concept of the Internet of Things. In chapter 6 we get to the heart of the IoT project in which we describe an experiment in an automated industrial warehouse scenario.

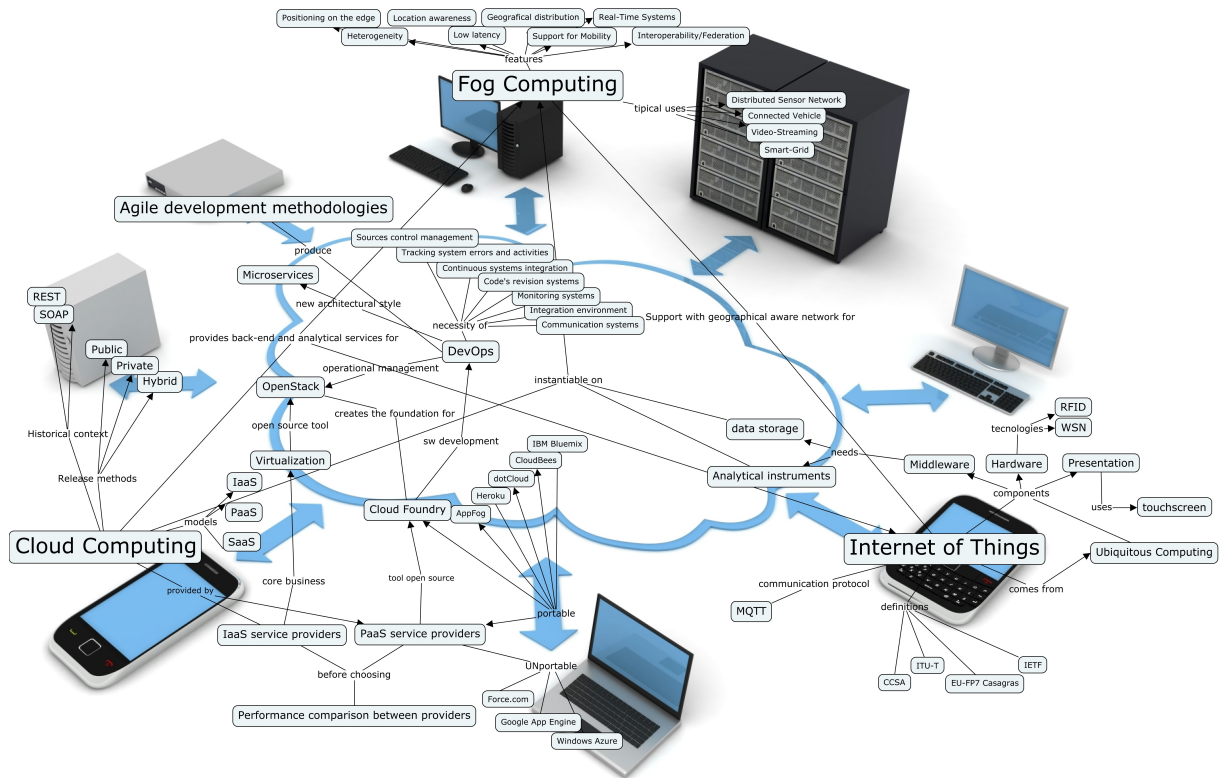


Figure 1.1: Conceptual map of the subjects covered.

Chapter 2

Cloud Computing

*“Computation may someday
be organized as a
public utility”.*
John McCarthy

2.1 Origin of the Cloud

John McCarthy, the researcher who coined the term *artificial intelligence*, spoke in 1961 of the possibility that the computation could have been provided one day as a public-access service [17].

In the year 2000 Roy T. Fielding coined the term REST (Representational State Transfer) [16] in his doctoral thesis to identify a software architecture particularly representative of the world wide web. Also he identified the constraints and the essential characteristics inherent in the network environment:

- **Client-Server:** separating the user interface from the data storage mode, it enables the user interface portability across multiple platforms and it improves scalability by simplifying the server components.
- **Statelessness:** each request from client to server must contain all information necessary to understand the request and can not

take advantage of any stored context on the server. The session state is then maintained entirely on the client. This constraint induces the properties of visibility, reliability, and scalability at the cost of a reduction in network performance, increasing the repetitive data (overhead for interaction) sent in a series of requests from the time that the data can not be stored on the server in a shared context.

- **Cache:** this constraint requires that the data within an answer are implicitly or explicitly labelled as stored in the cache or not. If a response contains the first label, the data may be introduced in the client cache so that the latter can reuse this data in response to the equivalent subsequent requests without the need to repeat the request. This mechanism partially compensates the overhead caused by the absence of state.
- **Uniform interface:** applying the principle of generality from software engineering at components' interface realization, the system architecture is simplified. Also visibility of interactions is improved. The implementations are decoupled from the services they provide, encouraging the possibility of independent development. The tradeoff is that a uniform interface decreases efficiency because informations are transferred in a standardized form rather than in a specific representation tailored for the application needs.
- **Layered system:** this stratification approach reveals an architecture composed of hierarchical levels constraining the visibility of the layer components immediately nearby with which they interact. By limiting the knowledge of a single-level system, it puts a limit to the overall complexity of the system and it promotes the independence of each substrate. The main disadvantage of layered systems is that they add overhead and latency in data processing, reducing user-perceived performance.

- **Code on-demand:** REST allows you to extend the capabilities of the client by downloading and executing code in the form of applets or scripts. This makes possible the simplification of the client by reducing the number of elements that must be released in advance. Allowing the code to be downloaded after deployment improves the extensibility of the system. However, it reduces the visibility.

Almost simultaneously began to spread concurrently around 2000 the *web services* paradigm [15], which proposed special attention to the common question of interfaces uniformity and proposed as standard a new language derived from XML, WSDL (Web Services Description Language), which made it possible to semantically describe the services provided to the client and everything you need to invoke them. The protocol that allowed the whole system to work was SOAP which reflect a service-oriented architecture based on three classes of interacting entities: **service requestor**, **service provider** and the **discovery agencies**. The benefits of Web services were then due to decoupling between services by the implementation of interfaces on every specific platform, making it possible to dynamically link services. They also helped achieving the interoperability between different languages and platforms.

Access to services were finally made public but the one to the computational resource not yet. In fact the computational resource was bounded within the service providers' server. Service providers was the exclusive makers of those services and they alone could chose which services to publish.

In this atmosphere of struggle between SOAP, service-oriented, and REST, focused on resources, we come to 2005, when *Amazon* [27] had already spent more than a decade and millions of dollars in building and managing an IT infrastructure on a large scale, reliable, efficient and that had supported one of the largest online sales platforms retail worldwide. How to use all this equipment during the summer? It re-

mained in almost total inactivity. Why not sublease these servers to try to recover the investment costs?

In 2006 it was thus launched Amazon Web Services (AWS) so that other organizations could benefit from Amazon's experience in the management of an IT infrastructure on a large scale, distributed and transactional.

AWS and Azure Services Platform from Microsoft have generally adopted the Web Service [17] based API in which, access to users, Cloud configuration and services used the default API presented as Web services. The selected protocol was thus SOAP.

However it is clear that the *cloud computing*, as currently understood comes from more recent origin. The First International Conference on Cloud Computing took place only in 2008 and another international conference on cloud computing met in December 2009.

In general, the *Cloud* as commonly understood, provides services at three different levels (IaaS, PaaS and SaaS) and some providers may even choose to provide services to multiple levels.

2.1.1 SOAP vs REST: final considerations

In the early implementations of service-oriented architecture, standards such as SOAP, XML-RPC, and WSDL were common for the structuring of the data and in communication via API. These standards, however, were heavy and inflexible [10] and were generally quite difficult to use. In the re-emergence of services-oriented architecture today there are new concepts, more agile and light as REST [1], which integrates agility and speed, while maintaining the advantages of a distributed system with weak coupling. According to these considerations Amazon has decided to support either SOAP and REST API for public access to their cloud and found that 85% of customers opted for the REST alternative. The reasons for this preference can be explained by the fact that REST solutions has already become pervasive. HTTP clients and servers are available for all major programming languages and the default HTTP

port (80) is commonly left open as standard setting in most firewall configurations. An infrastructure so light [23], where services can be built with minimal tools, is cheap to acquire and therefore has a very low barrier for adoption. The effort necessary to build a REST client is very low, also the developers can begin to test these services from a standard web browser, without the need to develop any specific client-side software. Distribution of a RESTful Web service is very similar to building a dynamic website. Moreover, thanks to URIs and hyperlinks, the REST architecture has demonstrated the ability to discover web resources without a dependent approach implying a mandatory registration to a central repository. From an operational point of view, it knows how to scale up and down a RESTful Web service in order to serve a very large number of customers with caching support, clustering and load balancing implemented in REST. The REST adoption removes the need to perform a number of additional architectural decisions related to the various layers of the Web Service and make it seems such a complexity superfluous. Although Web services based on SOAP remain a feasible technology within the big industry when you have advanced QoS requirements, REST has become the tactical alternative for systems and distributed services integration paving the way for public access to computing invoked by John McCarty. We will see such an architecture applied in the paragraph 3.4.2 about portable Cloud.

2.2 Infrastructure as a Service (IaaS)

The paradigm "infrastructure as a service" (IaaS) offers hardware, software and equipment to provide customers with application environments with a pricing model based on resources utilization. These infrastructures are able to scale in both directions dynamically as resource are needed by applications, avoiding users to manually configure bandwidth, memory and storage statically. Vendors may compete on performance and prices offered for their dynamic services. The service provider is the hardware owner and is responsible for the management

and maintenance of the servers. The IaaS services are generally purchased on a pay-as-you-go base because, the chance of paying only for the resources required enables the possibility of converting capital expenditures into operating expenses.

Using this technique [7], virtual machines are created on premise and loaded with the software to run on the cloud. Then the virtual machine must be managed by the customer and configured to the hosting environment of the IaaS provider to use the storage services provided. It is customer's responsibility to also monitor all the software running on vm including license management. IaaS use option is very flexible but its adoption is only recommended if the application migration to the cloud must be very fast and there is no time to re-engineer the code for the new environment.

Typical examples are *Amazon EC2* (Elastic Cloud Computing) [27] and *S3* (Simple Storage Service) [1, 27] in which the entire calculator and the data storage infrastructure are accessible to customers.

2.3 Platform as a Service (PaaS)

The "Platform as a Service" (PaaS) provides an integrated environment of high level to build, test and deploy ad hoc applications. In general, developers will have to accept some restriction on the type of software that they can write. In exchange for this sacrifice they can obtain scalability for their applications managed as a built-in service offered by the provider. An example is the Google App Engine [24], which allows users to build web applications on the same scalable systems that support Google apps.

The PaaS allows you to take the best practices without thinking about it [10]. From a developer's point of view, shared web hosting is easy, but it does not provide control and power, while the dedicated hosting is quite powerful, but involves too many distractions and soft skills. Until the advent of PaaS were never existed a middle ground that could provide the power, the speed, reliability and scalability that you wanted

with the dedicated hosting nevertheless remaining simple to use as the shared hosting. The reliability and scalability are made possible by the N-levels architecture which is one of the fundamental principles of REST.

We will enter in deployment's details of this service model in chapter 3.

2.4 Software as a Service (SaaS)

The model "software as a service" (SaaS) offers a specific sw product accessible to customers remotely through the internet with a pricing model based on usage. SaaS helps organizations avoiding capital investments and lets them focus on their core business rather than on support services such as IT, infrastructure management, software maintenance, etc. In addition, by removing the dependency on local product installations, SaaS provides access to applications worldwide. To accommodate a large number of cloud users, SaaS applications can have multiple users simultaneously so any cloud machine may serve users from different organizations.

Fit into this category Salesforce's products. Salesforce is one of the leading provider of *on-line CRM* (Customer Relationship Management). Another example is Microsoft's *Live Mesh* that allows you to share files and folders in addition to the synchronization of multiple devices.

2.5 Cloud types

The cloud computing services previously described may be released in different forms, each of which involves different levels of safety and need for maintenance.

2.5.1 Public Cloud

A cloud is told public when services are made available through an open network for public use. The cloud service providers such as Ama-

zon AWS, Microsoft and Google, own and operate only the data centres. The access is typically via the internet without direct connectivity possibility. From a safety point of view we must consider the substantial scenario diversity for these services (applications, storage, etc.) because they are made available by a service provider through a public net. In this scenario the communication is done through a not fully reliable network which requires a deeper attention to security requirements.

Regarding specifically the public cloud *Platform as a Service* [10], usually you can find it on a public *Infrastructure as a Service* platform (such as Amazon Web Services [27]). This is for example the case of PaaS providers like Heroku, EngineYard and AppFog. In many PaaS options you are not given the chance to choose exactly where the code is executed. You do not have much control over what is happening in the service, nor you are allowed to work on the operating system mechanisms. The customer provides the code and PaaS runs it with the disadvantage of loss of understanding of what actually happens in the server.

2.5.2 Private Cloud

The term private cloud identifies a cloud infrastructure in which the ownership is entirely recognizable in a single organization. Take a private cloud project requires a high level of commitment by the organization to virtualize the environment in which it operates its business. The enterprises are also required to check the allocation of existing resources. A private cloud can increase their chances of business by opening the offer to new markets, but every step of the project can raise security issues that must be addressed to prevent serious vulnerabilities. The construction of autonomous data-centers generally involves large capital investment and require the ability to manage a large physical place for hardware allocation and rooms' cooling. The cloud infrastructure must also be updated periodically resulting in an

additional investment of capital. Following these considerations, we can say that private cloud removes those economic benefits of cloud computing that make it such an interesting alternative.

The private PaaS cloud is much less familiar to most developers than its public counterpart [10] since it can take many different forms, but in a nutshell it is PaaS running on proprietary hardware. It can be executed on an on-premise IaaS platform type, such as OpenStack, vSphere, CloudStack, Eucalyptus or even directly on unvirtualized hardware. The difference is usually that in private cloud the owner/developer has responsibility for code's management. Those who run a private cloud, as opposed to those who rely on public cloud PaaS, get similar functionality and the same release mechanisms for the execution of applications, but are also responsible for code's behavior on the PaaS and they will have to worry about its running state. This way can offer more control over servers and the ability to use proprietary hardware without the need of being tied to a particular service provider.

2.5.3 Hybrid Cloud

The hybrid cloud consists of an integration of different public and/or private clouds which remain separate entities but are unified and allow the integration of services provided by different providers. Some vendors, such as AppFog and OpenShift, provides the possibility for a system to be extended over public and private clouds, in fact these providers allow you to choose where do you want each application to be hosted.

A promising frontier for the hybrid cloud contemplates its use by the public administration. Consider a common scenario where you're building an application that uses sensitive data with the limitation of having to maintain them internally. This part of data with restrictions usually is only a small portion of the total system but in a monolithic application it forces the entire system to be executed internally due to the constraints on sensitive data. However, if you design the software tak-

ing in consideration a distributed execution, using a lot of API to allow a lightweight frontend and decomposing the application into multiple services, it is still possible to allocate most of the system safely on a public cloud. All components that don't interact with sensitive data can be performed on external hardware lowering the costs of the ICT infrastructure. This approach, which involves the construction of many small services works particularly well with the PaaS paradigm. The realization of small independent services to be connected together is not only a modern approach to the development of applications for both smart devices and web, but it is also particularly suitable when you're considering how to build a PaaS strategy that combines the services of multiple public clouds with a private cloud in which to keep sensitive data.

Chapter 3

Cloud service providers and software development

PaaS service providers can choose to adopt very different infrastructures, but in general they maintain the virtualization software that runs on proprietary hardware, directly on the server they own or acquiring the services of an external IaaS provider. In the first case the PaaS service provider must also take care to maintain the IaaS level on-premise using OpenStack, vSphere, CloudStack and Eucalyptus or even directly if the hardware is not virtualized.

The number of new PaaS service providers is increasing rapidly. As shown in chapter 2, the cloud can represent a unique opportunity for software developers but, to ensure that the choice of the service provider is prudent and conscientious, it is essential to investigate how the underlying virtualization technologies work. We will also consider the bounds between the application that must be developed and the provider's infrastructure. Finally, some general techniques for comparing different service providers will be presented.

3.1 Virtualization

PaaS provider that operates on proprietary hardware will also have the task of implementing an appropriate virtualization strategy otherwise

adopted by IaaS provider. The cloud architecture leverages virtualization techniques that can provision multiple virtual machines (VM) on the same physical host in order to make efficient use of available resources [13], for example, allocating the VM in the minimum possible number of physical servers reduces the energy consumption at runtime. For example, we can have two physical servers running a VM each and both are not using their full computing capacity. Therefore, the assignment of both virtual machines at the same physical server can usefully lead to the switching off of a machine. We should also remember that the energy savings will always be more effective in large data centers. Allocating VMs', however, we must think carefully about the aggregated resource consumption of virtual machines co-allocated in order to avoid a failure in providing the service level agreement (SLA) purchased by the customer. Thus the VM allocation also leads to several management problems because it requires an optimal usage of the available resources in order to avoid performance degradations caused by resource consumption of virtual machines co-allocated on the same host. In order to better address the issue of physical resources optimization (CPU, RAM and persistent storage systems) several algorithms have been proposed to solve the problem of allocating VM in order to improve load balancing among servers and minimizing the number of switched on machines.

3.1.1 Openstack

The proliferation of cloud service providers to which we are witnessing has had its start when the project **OpenStack** were opened to the public becoming open-source. This project provided a first implementation of a virtualization infrastructure that used the optimizing algorithms described above. So OpenStack is an open-source solution for the creation and management of cloud infrastructure (IaaS layer), originally developed by NASA and Rackspace. OpenStack also allowed small businesses to deploy infrastructure in the cloud even if its adaptation

to specific scenarios may require a deep infrastructure management re-engineering. OpenStack uses open-source libraries and components well known; It manages both the computing that the cloud storage resources to enable dynamic allocation of virtual machines. OpenStack is the result of the integration of two different important projects: the first, made by NASA and called *Nova*, mainly manages computing and network resources, while the second, provided by Rackspace, is called *Swift* and it is responsible for archiving files on a cloud platform. Early on, there were many criticisms about the immaturity of the software [10]. However it has matured rapidly and is becoming every day more and more stable and feature rich. In fact, the open-source nature of the project has enabled hundreds of companies large and small to invest time, money and resources to make OpenStack such a huge success that there are now at least three large public cloud based on it: one from Rackspace, IBM's Bluemix and another one from HP. Countless other initiatives are added each day proving the pervasiveness that this software solution has now reached.

3.1.2 Cloud Foundry

From level IaaS, the abstraction gap to achieve the services to be made available to the whole world is still high. To bridge this gap is essential to establish a new layer equipping a PaaS platform like *Cloud Foundry* that is releasable directly above OpenStack infrastructure described above (or on AWS or vSphere). Cloud Foundry [2] is an open source platform supported by a large community. The openness and extensibility of the platform prevents its users to remain confined to certain languages or at some set of application services. Cloud Foundry also helps to reduce the cost and complexity of ICT infrastructure configuration. Developers can distribute their applications on Cloud Foundry using the most common tools and without having to change their code. The main components that constitute Cloud Foundry are:

- **BOSH**: creates and distributes virtual machines (VM) on top of

a physical computing infrastructure. It implements and manages Cloud Foundry on IaaS infrastructure. To configure distribution, BOSH follows the configured instructions written in a manifest.

- **Cloud Controller:** it manages applications and other processes on the cloud's VM. It cares about the resource balance according to the demand and manages the application life cycle.
- **(Go)router:** it establishes the traffic routes incoming from the wide area network (WAN) to the virtual machines that run the applications required from outside. Usually it works together with a load-balancing system provided by the customer.

These Cloud Foundry's components communicate among themselves posting internal messages and using transmission protocols as HTTP and HTTPS, in addition to sending NATS messages between themselves in a direct manner. Cloud Foundry distinguishes two types of virtual machines: the VMs that form the platform's infrastructure and the VMs running applications as a host for the outside world. Within Cloud Foundry, a component called Diego distributes the hosted applications load on all virtual machines available, keeps running them and it manages balance issues for peak demand. Diego also manages interruptions or other changes in the network topology using an auction algorithm. Cloud Foundry uses the *GitHub* git system for the management of code version control and documentation. Developers working on the platform can also use GitHub for their applications, custom configurations and to manage other resources. To store large files in binary form, Cloud Foundry maintains an internal Blob archive. For storing and sharing temporary information, such as the states of the internal components, Cloud Foundry uses "Consul" and "etcd" distribution of values storage systems.

In general we can say that PaaS is the place of Software Engineering because at that right level of abstraction you can easily manage issues relating to the systems scale. At PaaS level you can accomplish and

solve problems that, if taken directly into production as in the past, lead to considerable efforts in order to support migration that lasted for months.

3.2 DevOps

We have so far discussed about the operational tools that allow you to build in a short amount of time a functioning cloud infrastructure using OpenStack, described in subsection 3.1.1, and Cloud Foundry in subsection 3.1.2, but how to integrate the operational management of these systems with the process of agile software development?

The most advanced software development team are trying to adopt semi-autonomous technologies to help manage the multitude of data and communication activities required to support the phases of the classical software life cycle. These tools send messages and collect/process data and they are able to show the important informations to the interested developers involved in the project so that they can take part to the appropriate stage of the software life cycle. These tools can perform complex tasks ideal for the automation, thus reducing the burden of distraction and the number of activities which affect humans.

The process of software development requires a large number of tools and information systems to manage data and processes. As the agile automation technologies become more capable and essential to the development process, their management and maintenance becomes increasingly complex for many teams. In many organizations, while the development team involved in the software production process, other operations teams manage the assistance tools and technologies required. This separation between different kind of specialists groups has resulted in a difficulty of communication between those involved in development and those that supports the operations. These issues were traditionally addressed by organizations that imposed priorities and direction of efforts for the maintenance and observance of the software engineering *best practices*.

The current practice of development has led to a new concept, called **DevOps** [12], describing the conceptual and operational fusion of software development (Dev) with operations (Ops) needed for the technologies to work properly. The key point of this philosophy is the integration of the operations teams, which support the development process and often the testing and release of software products, with the development teams who design and implement products. This fusion has the aim to maximize the utility of the essential development tools and, at the same time, align the priorities of the development team with those of the operations staff in order to promote a successful cooperation and to work towards the shared goal that the project provides. The organizational conceptual unity inherent in DevOps paradigm is naturally extended to the interoperability among operational and development tools with the aim of ensure maximum access to the data, the dissemination of knowledge and automation. DevOps achieves its goals partially replacing explicit coordination with implicit mechanism [19].

3.2.1 DevOps best practices

Trying to achieve DevOps in an organization there are some best practices that could arise minor or major architectural refactoring in the software design [19]:

- Treat Ops as first-class citizens from the requirements point of view. Adding requirements to a system from Ops may need some minor architectural change. The Ops requirements are likely to be in the area of logging, monitoring and information to support accident handling.
- Make Dev more responsible for relevant accident handling. By itself, this change is just a process change and should require no architectural modifications. However once Dev becomes aware of the requirements some architectural modifications may result.

- Enforce deployment process used by all, including Dev and Ops personnel. In general, when a process becomes enforced, some individuals may be required to change their normal operating procedures and, possibly, the structure of the systems on which they work. One point where a deployment process could be enforced is in the initiation phase of each system.
- Use continuous deployment. Continuous deployment is the practice that leads to the most far-reaching architectural modifications. In order for an organization to maintain continuous deployment practices with little effort, a major architectural refactoring is required and that we will bring us to deepen such an architecture in the next section 3.3 about *microservices*.
- Develop infrastructure code with the same set of practices as application code. These practices will not affect the application code but may affect architecture of the infrastructure code.

3.2.2 DevOps system components

When an automatic system is inserted in the process of software development communication activities to be carried out increase in complexity. The data and relevant information are stored both by human actors and the system. Since the man-machine communication does not occur through the natural language expressions, the use of previously defined software interfaces is needed. Furthermore the effectiveness of the whole operation requires that humans do not exchange messages between the system entities. The entities should be designed and arranged as to be able to pass information between themselves and as to present to humans the priority knowledge only when it is necessary, if possible, within the normal flow of work, reducing the effort needed to extract new knowledge from the system.

A DevOps system therefore will need the following components [12]:

- **Source control system:** the storage system and the file version

control sources and other artifacts necessary for running the software system such as the file with the configuration parameters and media files.

- **Tracking system for errors and activities:** a system to manage project activities with their status.
- **Build continuous integration systems:** systems able to compile, build and test the source code to produce a working application. The continuous building process with the ability to test the application each time you make changes to the source code is represented by the acronym CI which stands for *Continuous Integration*.
- **Documentation systems:** systems used to create, store, transfer and display the documentation relating to the project software. They are often manual systems that require the use of a text editor and e-mail to create and transfer documents among human actors.
- **Code's revision systems:** systems which review the software source code to ensure the accuracy and quality. Alternatively, these tools can also be used to help rewriting and making changes to sources by qualified human actors.
- **Monitoring systems:** systems that monitor the status and functionality of all the other systems to ensure proper operation and quickly inform the appropriate subjects in case of need.
- **Integration environment:** this is the environment in which all systems operate, both DevOps and the others systems made by the software artifacts produced during the projects development. Often it is a virtual infrastructure that allows the creation of virtual machines and their dynamic management.
- **Communication systems:** systems responsible for the communication of knowledge to human beings, both from other humans that from system software entities.

3.3 Microservice Architecture

A *microservice* [26] is a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility. It is a single responsibility in the original sense that it's got a single reason to change and/or a single reason to be replaced. But the other axis is a single responsibility in the sense that it does only one thing and one thing alone and can be easily understood. Most of the organizations have actually started with some big monolithic system and have split that big thing up after migrating to Cloud. That's the case for most organizations that are adopting a microservice architecture as, for example, Netflix.

So *Microservice Architecture* is an architectural style that satisfies three Cloud's requirements [19]:

- Deploying without the necessity of explicit coordination with other teams reduces the time required to place a component into production.
- Allowing for different versions of the same service to be simultaneously in production leads to different team members deploying without coordination with other members of their team.
- Rolling back a deployment in the event of errors allows for various forms of live testing.

Using this architecture, an application can now be seen as a composition of multiple services where each service provides a small amount of functionality that could even be a single functional requirement. The reason for this architecture to be so popular in DevOps ruled organizations is that a small team could be accounted for the entire lifecycle of a service. Each team has the ability to deploy their service independently from other teams, to have multiple versions of a service in production simultaneously and to roll back to a prior version relatively easily.

3.4 Code portability between providers

The possibility to move your application from a PaaS service provider to another is a variable to be examined carefully. The reasons affecting such a choice may arise from changes in economic conditions that affect the pricing of services provided, or the availability of a new best rated provider for costs and / or services. Portability or interoperability of systems across multiple cloud, is also a strategically important property to ensure the availability of software products released on PaaS.

This is particularly evident by examining [4] what happened to an on-line storage service called *The LinkUp* ended August 8, 2008 after losing 45% of the customers data. The Linkup, in turn, had taken advantage of another on-line storage service called *Nirvanix* to store data. This led to a conflict for the allocation of responsibilities between the two organizations trying to explain why customer's data had been lost. Meanwhile, 20,000 users of LinkUp was told that the service were no longer available and they have been asked to find a different provider. Another story that shows us the importance of implementing systems not strictly tied to a single service provider concerns Amazon and is inherent to the transfer of legal liability. The cloud computing services providers would like the legal liabilities associated with the customers who deploy applications on their systems introducing a separation of liabilities between applications running and Amazon's infrastructure. In fact in 2009, FBI raided a Dallas data center because a company whose services had been hosted by Amazon was being investigated for spam. However a number of systems housed in the same structure have suffered days of unexpected downtime and some have been forced to withdraw from the market.

From these stories we can learn how the release of systems on multiple cloud platforms simultaneously is a winning strategy that ensures the continuity of the business in the form of availability of services in addition to providing the ability to obtain an accurate comparison of the costs relating to the various providers.

The first providers of PaaS services were very restrictive with regard to the aspect of portability. The limitation was derived, as in the case of *Google App Engine* [24], from restrictions imposed on development languages. Such limitations, as the inability to create a new file or to use java socket in Google's cloud, were mitigated by the availability of APIs that allow a developer application to interact with the services of *Google App Engine*. In addition to the rapid learning curve that these APIs imply, the application that uses them is closely tied to the provider. Thus we can distinguish two categories of service providers: [10] portable and unportable.

3.4.1 Unportable PaaS

When a PaaS is not portable, you must create an application by writing code around specific platform APIs. This means that the code's structure must strictly embrace a specific model. The APIs may be centered on the database service, the storage mechanisms or search tools. In other cases, the APIs involve lower level issues related to the code. Sometimes you must also use specific languages specifically built for that platform.

Currently fall within the category of unportable PaaS the following providers [10]:

- **Force.com:** developed by Salesforce and launched in 2008. Designed to extend the services and improve the access to the corporate data. It is one of the first PaaS examples and was considered an inspiration for subsequent infrastructures.
- **Google App Engine:** also released in 2008, promises its users the opportunity to draw upon the potential of Google's infrastructures in addition to their experience in the field of systems management. To enable scalability, the application must strictly adhere to certain standards that have been identified taking into account the specific way in which Google's infrastructure operates and runs.

The application, being built around this agreement, it is guaranteed to operate at Google's horizontal scale, thus being able to handle a variable number of requests. But on the other side the standard imposes limitations on the file system access in addition to the time in which the application must respond. The latter limitation may arise vertical scaling problems, that is when the requests complexity grows.

- **Windows Azure:** this PaaS system was developed by Microsoft around the .NET framework and also released in 2008. Azure provides the developer with some libraries to access the services. Through these libraries Azure can operate the application scalability in a manner transparent to the developer's point of view. Other standard services provided by the cloud scale independently instead. With Windows Azure are provided basic systems, as a bus of messages and a queue system, in addition to a variety of different options based on the specific application needs, thus providing developers the patterns that help them in the construction of distributed applications that can interact with each other through the network.

Even if all these PaaS were initially classified as unportable, many of them are adding features that will make them more portable. GAE has recently taken steps to support the PHP, requiring fewer changes to the software to be integrated with Google's systems. Windows Azure also has released support for PHP developers leaving more and more autonomy to operate without the Microsoft API.

3.4.2 Portable PaaS

A portable PaaS is a platform built in such a way that it can run the code without requiring significant changes. For developers who have created the code for a shared or dedicated hosting environment, move the code in a portable Platform-as-a-Service shouldn't be difficult. There

are no services or API that must be absolutely used in order to make applications executable. The cloud services, if any, may be used freely. Portability expands the amount and type of code that you can write in the Platform-as-a-Service paradigm. With the extension of support for different languages, greater flexibility is allowed. If you want to move an application between different portable PaaS platforms, you only need to change little aspects of the application, but in general, these changes will not involve a complete rewrite of the system.

Currently fall into this category, the following providers [3, 10]:

- **Heroku:** it is one of the first portable PaaS services. Launched in 2007, it has been able to evaluate the projects that were carried out by Google and Microsoft and its developers have decided to diversify their system in order to become more open and suitable to host general purpose code that doesn't have the need to accede to specific APIs. It uses a git based code release mechanism.
- **AppFog:** entirely built on top of AWS (Amazon Web Services), it is a PaaS managed service. This means that users do not have to worry about configuring the platform. The main focus of AppFog is interoperability among different clouds. Application launched on AppFog can safely make use of services from other cloud infrastructure providers.
- **dotCloud:** it is an example of Platform-as-a-Service that has innovated to be the first supporter of multiple languages and technologies. It has made popular the idea of Linux containers making use of an open source project called Docker. This very popular PaaS is focused on creating a system that work from command line. Therefore it has a Unix shell and an API to interact with it, thus providing the ability to deploy applications in different languages.
- **CloudBees:** is a Platform-as-a-Service that focuses in particular on Java technology. It was built around the Java toolsets, and incorporates the most common components used within Java plat-

forms. One aspect that separates CloudBees from other cloud is its integration with tools like Jenkins to support continuous integration. In fact, CloudBees has assumed some of the people who maintained Jenkins and has thus become a leader in the development of continuous integration spaces. This PaaS provided a new breakthrough since it enables systems to extend the vision of PaaS. With other platforms, the common idea is to take the code and deploy it into production. CloudBees instead integrates multiple development tools to extend the knowledge it provides. Instead of just take the code and put it into production, CloudBees provides a system that allows you to test the code continuously, making sure that it works before going into production. It provides a very long pipeline before the code is deployed.

- **Cloud Foundry:** It is a recent technology developed by VMware to support the PaaS and enable its users to create their own private PaaS platform. It is an open source project very different from other cloud saw before as it leaves the PaaS configuration in the user's hands. Paradoxically it may even be run on a laptop. A major innovation is the generalization of the concept of service because, with its elevation at first-level abstraction, it simplifies connection and disconnection of services with the applications that use them. Cloud Foundry libraries provide many of the features you would expect with PaaS: to be able to deploy applications with a command line through REST API, scaling and load balancing tied to an application without having to configure anything manually and adding caching and database services such as MySQL and Redis. Cloud Foundry can instantiate and configure these components quickly and easily, but in case of failure, you must be operatively informed on how to debug Cloud Foundry to find the problem. The manager of the infrastructure also has the task of managing and scaling MySQL in order to meet the application requirements. The main components of Cloud Foundry are

described in section 3.1.

- **IBM's Bluemix:** [3] it is the Platform-as-a-Service solution from IBM that is based on Cloud Foundry and Docker. It uses a component called Devops as a tool with which the platform aims to make easy the development and release of both web and mobile applications using the abstraction layer of the infrastructure which is managed by SoftLayer. Bluemix allows you to develop, run, release into the environment and manage Cloud applications quickly, without having to deal with the creation and maintenance of the physical or virtual machines, network management, maintenance on the machines, the installation or update of the operating system, the database manager to store information, etc. Moreover, thanks to Devops, it allows the quick release of software solutions with new features in each machine where this cloud is running. Bluemix supports various programming languages (Java, Node.js, Go, PHP, Python, Ruby on Rails) and offers ready to use services for database management, reporting, Internet of Things, mobile applications etc. If you use any of the programming languages supported, Bluemix also provides its buildpacks, which is a set of scripts needed to prepare the code to run on the cloud. If you are interested in writing code in some other programming languages, Bluemix makes it possible through the creation and use of a specific buildpack.

With a portable system of Platform-as-a-Service, the great advantage is that you can take the existing code and distribute it more easily, without major rewrites. It can iterate faster. If you have the need to move an application from a particular system in another environment, usually it requires little effort. The advantages of a unportable platform instead rely heavily on the services that the cloud provides. For example, Google App Engine, the advantage is to connect your application to Google's infrastructure and operations. In the case of Windows Azure, the advantage inherent in the connection to the Microsoft operations.

The trade-off depends on the type of application you want to develop. For example, if you need to run an application in Node.js, you will not be able to do it on Google App Engine. But if you want to try Google services you will not be able to do it on Heroku and AppFog. The choosing of a portable or unportable PaaS depends on the needs of the project and the feature set that you need.

3.5 Providers comparison

The variety of cloud providers leads to a practical dilemma: what are the performance of a cloud provider compared to other suppliers? For a potential customer, the answer can help in choosing a provider that best suits his needs in terms of performance and cost. For example, you might choose a provider for applications that require heavy memory usage and another for applications with high computational requirements. The challenge is that each vendor has its own idiosyncratic ways to implement systems, in order to find a common ground between the various providers we must first conduct an analysis to characterize the performance of an IaaS provider. A cost / performance analysis among PaaS service providers is more difficult because of the increased complexity due to the elevation of abstraction. The elements of comparison, that is the most common services, which let us make a cost comparison are [20]:

- **Elasticity of the computing cluster:** a computing cluster includes virtual instances that host and manage the customer's application code. Between different suppliers, virtual instances are differentiated by the underlying server hardware, the adopted virtualization technology and hosting environment. Even within the infrastructure of the same provider there are a variety of levels of virtual instances available, each of them with a different configuration. The computing cluster is also "elastic," in the sense that a customer can dynamically scale in both directions for each

of the instances uses in order to cope with the variable workload of its application. Currently there are two mechanisms that can be used: the opaque method and the transparent one. The first requires that the customer in person to manually change the number of instances specifying a resize policy (AWS [27], Azure, and CloudServers), such as creating a new instance when the average CPU usage exceeds 60% . The second method instead automatically compensates for the number of instances without the intervention of the customer (GoogleAppEngine [24]). The main parameters adoptable to make a comparison between providers are: time to complete a same operation, cost and latency measure introduced by changing the scale.

- **Persistent data storage service:** in order to improve the scalability and the availability of a system, the cloud providers offer persistent storage to maintain the state of the application and data. Today there are three kinds of services for storage needs: **table**, **blob** and **messages queue**. The use of tables, similar to hash or a set of key value pairs, is designed for storing small files but whose access has to be very quick. The blob is designed to store large unstructured multimedia files as binary objects. Finally, the messages queue implements a global communications infrastructure that connects all the different instances. Currently there are two pricing models for its storage: one based on CPU cycles needed to perform an operation, where the most expensive queries are the complex ones rather than the simple ones (AWS [27], GoogleAppEngine [24]); the other consists of a fixed fee per request regardless of the complexity of the request itself (Azure, CloudServers). The parameters useful to operate the comparison in this category are: the requests response time, time to the consistency and the cost for each operation.
- **Cloud internal network:** the cloud's internal network connects all instances purchased by a customer and the services together.

The network performance is vital for applications in a distributed system. The internal network of the data center often has very different properties than the external one. To compare the performance of the internal networks we can measure bandwidth and latency of the channels, using the TCP throughput as a measure of capacity since the TCP is the main type of traffic for applications. None of the providers currently rate the internal traffic of their data-center.

- **Wide-area network:** the WAN is defined as the collection of network paths between cloud's data center and external hosts on the internet. Many providers offer different physical places to host customer's applications. Requests from an end user can be served by an instance that run close to that position to reduce latency. As a metric we can compare the latency obtained from the WAN to a provider with the optimal value that we could get to reach that same position.

Whit these measurement methods is possible to obtain quantitative comparisons of different providers but you should always keep in mind also a qualitative classification of the cloud [10]:

- **Private/Public:** if it's working behind a firewall or on proprietary hardware are private but not otherwise, as discussed in section 2.5.
- **Portable/Unportable:** this classification is only for PaaS cloud. If you can easily move an application from a PaaS provider to another means that both providers are allowing code portability. A PaaS that can execute code at most unchanged from the original is portable. For example, if the PaaS supports PHP and WordPress can load an application without changing the code this will be a PaaS that allows portability otherwise, if a PaaS is bound to proprietary APIs that make it difficult to switch to other suppliers, it

is defined unportable. The current service providers according to this feature have been described in section 3.4.

- **Managed/Unmanaged:** the cloud falls in the first case if does not need to be maintained by the customer and is therefore used as a service. In the second case will be customer's responsibility the following topics: configuration, security management, maintenance, ordinary conduction and updates.

Chapter 4

Fog Computing

The paradigm of Cloud Computing is progressively replacing the old concepts of mainframe and back-end server, but how to extend the services offered by cloud at network's edges? Similar in some ways to the cloud thus comes the paradigm of *Fog Computing* [25] to provide data processing, storage capacity and application services to end users. Because of its unique proximity to the end-user, this paradigm is presented as a natural solution to the problem of the large bandwidth that would be necessary to connect a distributed sensor network (DSN) directly to the cloud. Another interesting application scenario comes from the geolocation support, which makes the Fog computing very attractive for IoT systems (Internet of Things), which we will investigate in detail in Chapter 5.

When the techniques and IoT devices will become increasingly integrated into people's lives, the current cloud paradigm is unlikely to meet the needs for the support of mobility, the location tracking and low latency. For these reasons the Fog arises as a model for the edge of the network in order to improve the quality of service (QoS) in an industrial automation scenarios, transport and networks of sensors / actuators. In addition, this new infrastructure natively supports the heterogeneity of the devices since it includes the user's device with their access points, routers and switches.

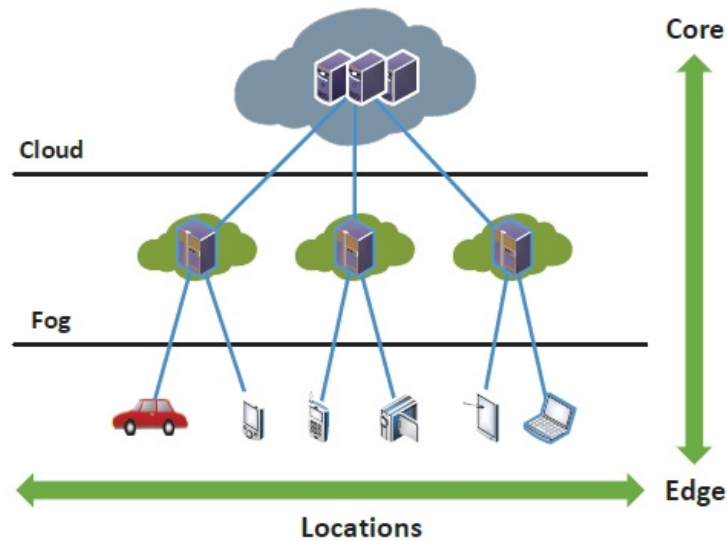


Figure 4.1: Fog between Cloud and network edge [25].

4.1 Fog systems features

Processing, storage and network resources are building blocks of both the Cloud and the Fog Computing. However, being at the network's edge involves a number of features that make the Fog a nontrivial extension of the Cloud [8]:

- **Heterogeneity:** it's a central element of the Fog model because nodes will be deployed in environments with very specific characteristics.
- **Positioning on the edge, location awareness e low latency:** the Fog origins can be traced back to the first proposals to support clients at the ends of the network with high information content services, including applications with low latency requirements.
- **Geographical distribution:** differently from Cloud centralism, services and applications, to which the Fog aims to support, operate in environments widely distributed as high-quality streaming for vehicles moving through proxy and access points located along

highways and race tracks. This brings as a consequence a high nodes cardinality and the predilection for wireless access.

- **Sensors networks on large-scale** : the Fog is ideal for environmental monitoring systems and for the Smart-Grid.
- **Support for mobility**: for many applications in the Fog field is essential the direct communication with mobile devices. Therefore it's important to provide mobility techniques that decouple the host's identity from its position. It is also required a distributed naming system.
- **Real Time interaction**: typical cases of Fog Computing involve the use of this paradigm in systems of autonomous vehicles.
- **Interoperability and federation**: full support to some services that require collaboration between different vendors. Consequently, the components of the Fog need to interoperate with services that are realized by connecting different domains.
- **Interconnection with the Cloud**: the Fog is well positioned to play a significant role in the acquisition and processing of data close to the source but you can easily make use of on-line analytical services that Cloud offers as outlined in section 3.

Chapter 5

Internet of Things

Thanks to the *Internet of Things* (IoT) paradigm [18] , many of the objects that surround us will be connected to the network in one form or another. Using technologies such as *Radio Frequency Identification* (RFID) and sensor networks, will be possible to respond to this new challenge through which ICT systems will be transformed by integrating the environment around us invisibly.

An indispensable component of IoT systems is the smart connectivity that will be made possible by context-aware networks such as those described in chapter 4. With the growing possibility of internet connection, mediated by wireless technologies WiFi and 4G -LTE, the access and the evolution to the ubiquitous information is already evident. However, in order for the IoT paradigm to emerge successfully, the today computing techniques must go beyond the traditional scenarios of mobile computing used by smartphones and laptops.

The cloud, as described in sections 2 and 3, remains the most promising alternative that can provide the backend infrastructure to support the most complex computations, for storing data and for the analysis tools. The cloud pricing model allows you to support end-to-end services for both businesses and users so that you can access required applications from anywhere.

In the IoT, the goal remains the connection of everyday objects so that intelligence can emerge from the environment around us.

5.1 IoT definitions

The term *Internet of Things* was coined by Kevin Ashton in 1999 within the framework of the customer-supplier chain management [5]. However, in recent years, the definition has been changed in order to achieve a more inclusive coverage that concern an extensive range of applications in health care, services and transports. Following this attempt to enlarge the concept of IoT in multidisciplinary environments we have seen the emergence of different definitions from many distant fields [11]:

- **IETF**: a wide global network of uniquely addressable interconnected objects based on standard communication protocols.
- **ITU-T**: a global infrastructure for the information society that enables advanced services through the interconnection (physical or virtual) of things based on existing and evolving information and communication technologies in an interoperable manner.
- **EU FP7 CASAGRAS**: a comprehensive infrastructure network that connects physical and virtual objects through the use of data acquisition and communication possibility.
- **CCSA**: a network that can collect information from the physical world, or control the objects of the physical world through various devices released with the ability of perception, computation, execution and communication. This network supports communication from human to "thing" or between things through the transmission, classification and processing of information.

Even if the definition of 'things' has changed and the technology has evolved, the main objective remains the same. In fact, the IoT is proposed to be able to automate the emergence of knowledge from the data received by a computer without human involved in the process. It is a radical evolution of the present Internet in a network that interconnects objects for gathering data from the environment, through the

sensing activities, and to interact with the physical world. It is intended to organize everything by using existing Internet standards.

5.2 Ubiquitous Computing

The *Ubiquitous Computing* is a discipline born in the '80s with the aim of creating an interface from human being to human being [18] in every day life through the use technology. The invention of the Internet has marked a first milestone realizing the vision of ubiquitous computing by allowing devices to communicate with any other device in the world. Since Mark Weiser [28] made explicit the ubicomp vision, a large research community able to address multidisciplinary aspects was founded. Several successful prototypes were built and evaluated showing the utility of Ubicomp systems in different fields. At the same time ICT technologies diffusion have made great progress by introducing low-cost solutions and enabling services that fulfill the ubicomp vision. Probably the biggest success of these products is the smartphone that has become part of the daily lives of billions of people. Smartphone diffusion is building an environment with increasing amounts of computational capabilities, detection and communication.

Caceres and Friday [9], discussing the progress, opportunities and challenges during the anniversary of the 20 years since the introduction of this discipline, identify the building blocks of the Ubiquitous Computing and the features necessary to maintain the discipline up with the times. In their analysis they identified two basic technologies for the growth of the Ubiquitous Computing: Cloud Computing infrastructure and the Internet of things. The cloud computing services for people constitute themselves as natural companions for personal mobile devices and for the future of ubicomp applications. The utility computing on the cloud can provide important back-end resources for ubicomp applications to be integrated with sensors networks and actuators in your environment as expected from the paradigm Internet of Things (IoT). Unlike Ubiquitous Computing, the IoT focuses on interaction between

real world objects instead and not among humans.

5.3 Constituent elements of the IoT

There are three components of the Internet of Things that also support ubicomp requirements [18]:

- **Hardware:** RFID technology is a major step forward in integrated communications systems because it allows the design of microchips equipped with wireless communication capabilities making possible the automatic identification as if the objects were equipped with an electronic bar code. Passive RFID tags are not powered by battery and use the power of the reader's interrogation signal to communicate their ID. In contrast, active RFID tags have their own battery power and can create a communication instance. Recent technological advances in low power integrated circuits and in wireless communications have made available miniaturized devices for the remote sensing applications that are efficient and low-cost. The combination of these factors has made it possible to use sensor networks formed by a large number of intelligent detectors that allow the collection, processing, analysis and dissemination of valuable information gathered in a variety of environments. Typically, a node of a wireless sensor network, contains interfaces for sensors, processing units, transmitter-receiver units and the power supply. Almost always they include also multiple A / D converters for the sensor interfacing. The most advanced models have the ability to communicate using an entire frequency band that makes them more versatile.
- **Middleware:** one of the most important consequences of this emerging field is the creation of an unprecedented amount of data. Storage, property and the expiry of the data becomes critical. Internet currently consumes about 5% of the total energy produced in one day and with the types of planned scenarios is sure that

the demand will rise even more. Therefore, it is strategic the use of energy-efficient data centers to ensure efficiency and reliability. The data must also be stored and used in an intelligent way so it is important the development of artificial intelligence algorithms that may be released in a centralized way, making use of cloud computing infrastructures described in chapters 2 and 3, or distributed, following the principles shown by Fog computing in chapter 4.

- **Presentation:** visualization and information interpretation through tools that can be widely available on different platforms and can be designed for different applications. Visualization is critical in a IoT application as it allows the users interaction with the environment. Thanks to the touch screen technology the use of tablets and smartphones has become very intuitive. So the average user can fully benefit from the IoT revolution only if the display is attractive and easy to understand. This can be accomplished by promoting the adoption of data conversion policies into knowledge (fundamental in faster decision making).

5.4 IoT focused on the Cloud

The concept of IoT can be seen from two different perspectives: a vision centered on the 'Internet' and one centered on the 'things' [18]. The architecture that unfolds starting from the Internet provides as main objective the connection of Internet services while data are provided by objects. Otherwise, in the architecture centered on the 'things' smart objects take the center of the stage. Focusing on the first vision, in order to realize ubicomp full potential, a picture that shows the cloud at the center appears to be the most practicable not only to promote the necessary flexibility to the subdivision of related costs in an optimal manner, but also to be supported by a highly scalable infrastructure. This scenario includes:

- **Detection services providers** able to connect to the network and upload their data using a cloud-based storage system.
- **Analytical tools developers** that can supply their products.
- **Artificial intelligence experts** that can provide data mining tools and machine learning useful in data processing into information and information into knowledge.
- **Computer graphics developers** able to offer a wide range of visualization instruments.

Cloud computing can offer these services on IaaS, PaaS or SaaS levels. Thanks to the Cloud the full potential of human creativity can be provided "as a service", as explained in chapter 2, remaining in agreement with the paradigm of Ubiquitous Computing by Weiser [28]. Reaching the full potential of the Internet of Things in various application domains, the generated data, the tools used and the visualization issues are made in overshadowed since the Cloud embodies all ubicomp purposes by providing scalable storage services, computation time and other tools to create new economic opportunities. In chapter ?? we will describe the development of a case study that take a simple IoT system as a data source generator that needs a cloud platform like IBM Bluemix for analytical purposes.

5.5 IoT focused on the Fog

Systems based on the Fog Computing [25] are becoming an important class of IoT and CPS (Cyber-physical Systems) systems. The CPS systems have a close correlation between computational and physical elements as well as coordinating the integration of computers and high information density systems. Both IoT and CPS promise to transform our world by setting up new connections between control and communication computer systems integrated with the physical reality. The Fog Computing, besides bringing benefits as described in chapter 4, in

this scenario is built around the concept of embedded system in which software applications and computational capabilities are built into the devices for a different reasons than the computational one. Examples of this type include toys, cars, medical devices and equipment aiming to integrate the abstractions and software accuracy with the dynamics, uncertainty and the noise of the physical environment. Using the knowledge, principles and methods that arise from the CPS, we will be able to develop new generations of devices and intelligent systems in the medical field, highways, buildings, factories, agricultural and robotic systems.

An application case which will become prominent in the near future is the one of the *connected vehicles* [8]. This scenario is rich in connectivity and in car to car interactions, between cars and access points and among the same access points. The Fog, as seen in section 4.1, shows a number of features that make this the ideal paradigm for providing services such as those required for street traffic management. For example an intelligent traffic light system may interact locally with a number of sensors capable of detecting the presence of pedestrians and cyclists in addition to measuring the distance and speed of vehicles approaching. In addition, an intelligent traffic junction will also interact with the traffic light neighboring nodes to coordinate the lighting of a green wave. Based on this information the traffic light can send warning signals to approaching vehicles as well as change its stroke to prevent accidents. Coordination with neighboring nodes will be mediated by the layer of Fog through which you can make any changes to the cycle of a traffic light on the network. The data collected by the sensors are processed to make real-time analysis and to change accordingly the timing of the cycles in response to traffic conditions. Grouped data from intelligent traffic lights will eventually be sent to the cloud for more complex analytical analysis on the long-term.

5.6 MQTT

To support communication in IoT environments is becoming more and more relevant the use of MQTT protocol [6], born in 1999 at Cirrus links and today the maintained by the OASIS consortium for open standards for which IBM is part.

MQTT is a messaging transport protocol Client / Server based on the pattern publish / subscribe. It is lightweight, open, simple and designed to be easy to implement. These features make it ideal for the many situations, including environments with constraints, as in the Machine to Machine communication (M2M) and in the Internet of Things (IoT) or contexts in which it is required the use of minimal overhead because the network bandwidth is a rare resource. The protocol runs over the TCP / IP or other network protocols provided that enable messages dissemination sorted, lossless and bidirectional.

The most popular use of this protocol currently is in the application *Facebook Messenger*, released for all mobile platforms. Facebook Messenger is an application for instant messaging services to communicate text and voice integrated with the web-based Facebook chat functionality.

5.6.1 MQTT features

The features of MQTT include:

- The use of the messaging pattern publish / subscribe that supports one-to-many distribution and allows the decoupling of applications.
- MQTT is a transport protocol for messaging that has no knowledge regarding the payload content.
- Three different levels of quality of service (QoS):
 - **At most once**, in which messages are delivered according to the best efforts of the operating environment. The message

loss can occur. This level could be used, for example, with the environmental sensor data when it does not matter if a single reading is lost.

- **At least once**, where it is ensured that the messages arrive but duplicates may occur.
 - **Exactly once**, where it is assured that the message gets exactly once. This level could be used, for example, with a counters in which the duplicates or lost messages can result in a divergence from the true value.
- A small transport overhead and exchange of messages for the protocol minimized to reduce network traffic.
 - A mechanism to notify interested entities in the event of anomalous disconnection.
 - MQTT makes use of the broker pattern for the dissemination of messages. The solutions available for this component are the following: ActiveMQ, Apollo, HiveMQ, IBM MessageSight, JoramMQ, Mosquitto, RabbitMQ, Solace Message Routers, and VerneMQ.

5.6.2 MQTT methods

MQTT defines methods or verbs the desired actions to be performed on the identified resource. What this resource represents and whether the data is dynamically generated or not, depends on the specific implementation and the IoT context. The methods, which recall the REST style and are fully compatible with this network architecture are:

- **Connect**: it waits until a connection is established with the server.
- **Disconnect**: waits for the MQTT client to stop all work currently undertaken and that the TCP / IP session is finished.
- **Subscribe**: wait for completion of the method that allows a client to subscribe to one or more topics.

- **UnSubscribe:** it asks the server for the elimination of the client from one or more topics.
- **Publish:** it returns to the application thread immediately after it submits the request to the MQTT client.

Chapter 6

Case Study

Taking the goal to prove that the above theories are well grounded into reality, we proceed illustrating a case study linking all the previous models. A common scenario of industrial warehouse will be here seen as the place in which to apply IoT concepts whit the perspective of extend the system with the support of *microservices* hosted onto a PaaS Cloud infrastructure.

6.1 Product requirement specification STEP 1

6.1.1 Business requirements

Develop a system able to manage a warehouse using autonomous tools and to run without oversight of human operators.

6.1.2 Architectural requirements

Design and build a prototype of a software system that can catches interactions between two kind of entities in an industrial warehouse scenario:

- **Robot:** is the autonomous active entity whose job is to move supplies among rooms.

- **Room:** is the smart-environment in which is possible to store supplies.

6.1.3 Scenario

The robot, moving along a corridor, is carrying a pallet. Each room can host up to two pallets. When engaging a room, the robot must understand the situation and chooses to deposit the pallet or move to the next room.

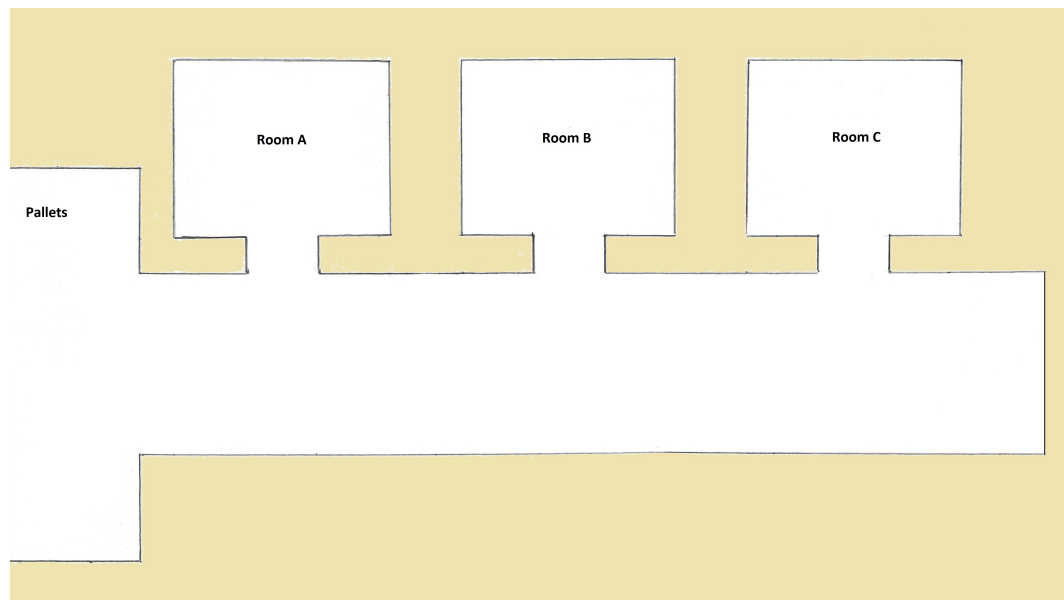


Figure 6.1: Environment map view from above.

6.1.4 Functional requirements

- A robot must be able to move through a differential drive system.
- Only one robot at the time is allowed to enter in a room.
- At boot time a room must retrieves his previous saved state over the net.
- A room can host up to two pallets.

6.1.5 Non functional requirements

Remember to express in explicit way the technological hypothesis assumed during the problem analysis phase, to define the abstraction gap (if any) and to explain how the software project can overcome (in a reproducible way) such a gap.

6.2 Product requirement specification STEP 2

Data analytics on the Cloud.

6.3 Analysis - STEP 1

To handle the complexity inherent in the analysis phase we will use a custom framework named QActors. We also operate under the technological assumption that we have a MQTT broker running outside the modelled system. Two different brokers have been used and tested: Mosquitto and Paho in its Eclipse implementation. These solutions are interchangeable at will. Every MQTT broker can only retain one message so it's important to take trace in a MQTT message of every previous transition of state occurred to the room.

When a robot enters in a room, many scenarios may occur. A precedent failure of the MQTT infrastructure could have lead to an inconsistency. So it's important to take in consideration every possible state of the world.

Position of failure:

	Robot	MQTT Broker
case A	Ok	Ok
case B	Ok	Fail
case C	Fail	Ok
case D	Fail	Fail

According to a different docking politic the results may diverge:

- **First a robot deposits its load then it sends the MQTT message:** in *case A* we have no inconsistency. In *case B* a fail has occurred in the server side and three scenarios now open according to the three levels of Quality of Service that MQTT offers. Whit level 1 (fire and forget) a robot has no chance to sense the fail. At level 2 (at least once) and level 3 (exactly once) the robot won't see any ACK coming from the server and will be stopped trying to contact the broker until a time out expired releasing the robot from its current task. In this scenario we have a divergence of "-1" in the IT model of the world. In *case C* and in *case D* the fail has occurred in the robot right after it has unloaded. Robot's inability to account the change lead to another divergence of "-1".
- **First a robot sends the MQTT message then it deposits its load:** in *case A* we have no inconsistency yet. In *case B* a fail has occurred in the server side and three scenarios now open according to the three levels of Quality of Service that MQTT offers. Whit level 1 (fire and forget) a robot has no chance to sense the fail so it will go on unloading the pallet and causing an inconsistency of "+1". At level 2 (at least once) and level 3 (exactly once) the robot won't see any ACK coming from the server and will be stopped trying to contact the broker until a time out expired releasing the robot from its current task. Meanwhile, with QoS level 2 and 3, the robot has not unloaded the pallet yet and can now

chose to abort the operation, move away with its pallet without harming the base of knowledge. In *case C* and in *case D* the fail has occurred in the robot right after it has sent the message to the MQTT broker. In *case C* we will have a divergence of "-1" while in *case D* we are relatively lucky and the faulty change won't be accounted due to the error state of the broker.

The probability of fails previously underlined lead to the best practice of checking the state of a room every time a robot step in taking a photo and comparing it with the value obtained from the broker. Possibility of casting an alarm should be taken into consideration at this point.

6.3.1 QActors

QActors (Quasi-Actors) is the name given to a custom framework built by Antonio Natali for the course Engineering of Software Systems to show how application designers can face the analysis, the design and the implementation of (distributed heterogeneous) software systems whose components interact by adopting a message-passing, an event-driven or an event-based style rather than a traditional object-based style. The QActor framework is inspired (with modifications) to the Actor model (Akka) and to the event-driven programming paradigm.

So, why using QActors?

QActors arise to tackle the issue of executable models definition helping designers and developers in contexts of distributed and heterogeneous systems such as the Internet of Things. QActors support the process of software production since from the problem's analysis phase and allow the rapid prototyping. This last feature is made possible through the technology *XText* [14]. *Xtext* is a framework that dramatically reduces the effort of building good tooling for a language. From a grammar, *Xtext* can generate a parser, a serializer and a smart editor. All concerns of *Xtext* itself and of the code generated by *Xtext* can be

customized via dependency injection. The kind of language that we are able to define can range from small Domain-Specific Languages (DSL) to full-blown General Purpose Languages (GPL).

A designers team defines a meta-model that XText can use to build a software factory. Then, models submitted to the software factory are made executables in a pure Model Driven Software Development (MDSD) fashion. The meta-model could be expressed in such a way to catch emerging IoT concepts like Microservices so that the software factory can automate the implementation even for different PaaS platforms allowing the deployment on multiple providers. Otherwise, the software factory product could be intended for the integration with an ensemble of DevOps tools.

The specific meta-model that gave birth to QActors is summarized in the following Ecore diagram:

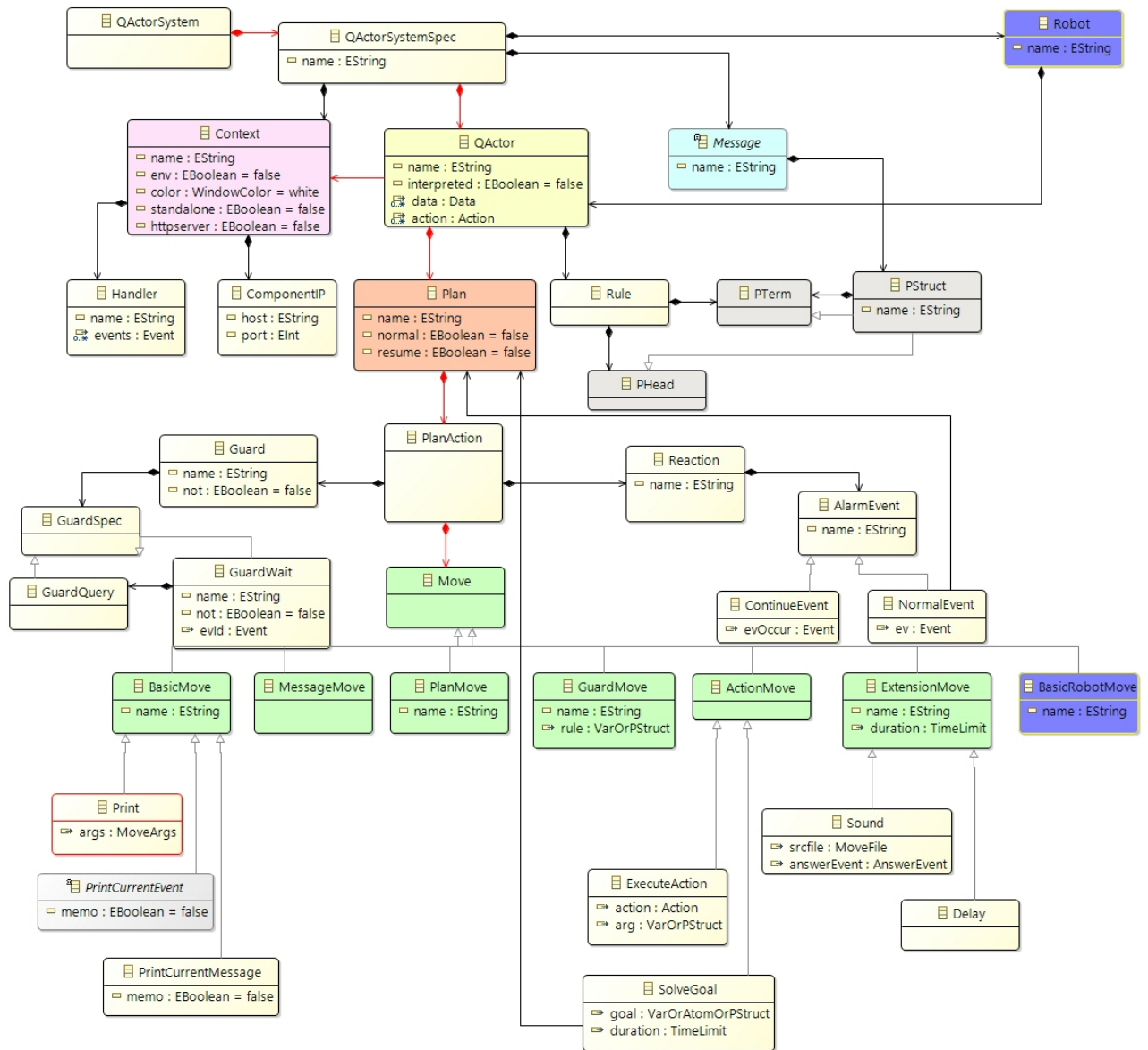


Figure 6.2: QActors DSL diagram.

The QActors framework promotes a precise process workflow based on a proper problem analysis.

6.3.2 Workflow for QActors - Phase 1

Find the main subsystems and define the System **Contexts**:

```
_____ System contexts identified _____  
  
context( ctxRobot, "localhost", "MQTT", "81" ).  
  
context( ctxRoomA, "localhost", "MQTT", "82" ).  
  
context( ctxRoomB, "localhost", "MQTT", "83" ).  
  
context( ctxRoomC, "localhost", "MQTT", "84" ).
```

6.3.3 Workflow for QActors - Phase 2

Find the main **applicative actors** working in each context:

```
_____ Applicative actors identified _____  
  
qactor( robot001, ctxRobot ).  
  
qactor( rooma, ctxRoomA ).  
  
qactor( roomb, ctxRoomB ).  
  
qactor( roomc, ctxRoomC ).
```

6.3.4 Workflow for QActors - Phase 3

Define the type of the **logical interaction** between actors by using the custom **high-level interaction-vocabulary**.

- *rooma* and every other room **connects** to the *iot broker* specifying the MQTT topic:

```
connect( "rooma", "tcp://m2m.eclipse.org:1883", "rooma" )
```

- Then, every room saves its own state **publishing** to the *iot broker* with QoS level 2 and the retain option enabled:

```
solve publish( "rooma", "tcp://m2m.eclipse.org:1883", "rooma", status(A,X) ,2,true)
```

- *robot001*, moving along a corridor, detects the presence of *rooma* through a rfid sensor and **reacts**.

```
solve actorOp(scanningforrooms)time(7000) react event roomdetected -> connecttoroom or event obstacledetected
```

- After having acquired the specific topic of the room through the active RFID info, *robot001* **connects** to the *iot broker* as a **subscriber** of room's topic:

```
solve connect( "robot001", "tcp://m2m.eclipse.org:1883", TOPIC ) time(0)
solve subscribe( "robot001", "tcp://m2m.eclipse.org:1883", TOPIC ) time(0)
```

- *robot001* receives the *iot broker's* reply and analyses the **dispatch** named **mqttmsg** discovering if the room is full or empty:

```
receiveTheMsg m (MSGID,MSGTIPE, SENDER ,robot001,mqttmsg( TOPIC,PAYLOAD ), MSGNUM)time(9000);
```

6.3.5 Workflow for QActors - Phase 4

We define the structure of the application **messages** exchanged by the actors:

```
Dispatch mqttmsg : mqttmsg( TOPIC, PAYLOAD )
PAYLOAD : status( TOPIC, X )
```

The PAYLOAD should be structured in such a way to carry the history of transitions happened.

Furthermore this is the very place in which to introduce security policies. Sending the PAYLOAD as plain-text exposes the system to security

vulnerabilities. To mitigate risks in an extremely open scenario, like the IoT one, we should consider encryption with asymmetrical keys kept privately by the publisher and the subscriber.

6.3.6 Workflow for QActors - Phase 5

Filling the gap between analysis and design steps, we specify the logical behaviour of each actor.

```

Robot logical behaviour
QActor robot001 context ctxRobot -g yellow {
  Rules{
    loadTheory(File) :- actorPrintln( loadTheory(File) ),consult( File ).
  }

  Plan init normal
    println(robot001( starts ) );
    solve loadTheory("./pahoTheory.pl") time(0) onFailSwitchTo prologFailure;

    addRule pallet;
    delay time (7000);

    //switchToPlan testConnection;
    switchToPlan search;
    switchToPlan roomInteraction;
    println(robot001( ends ) )

  Plan search
    println(robot001( searching ) );
    solve actorOp(scanningforrooms)time(7000) react event roomdetected
      -> connecttoroom or event obstacle detected -> obstacleavoidance

  Plan connecttoroom
    println(robot001("room detected") ) ;
    println(robot001("acquiring room info") ) ;
    [!? room(RFIDNAME,NAME,IP,TOPIC)] solve connect( "robot001",
      "tcp://m2m.eclipse.org:1883", TOPIC ) time(0) onFailSwitchTo
      prologFailure;
    [!? room(RFIDNAME,NAME,IP,TOPIC)] solve subscribe( "robot001",
      "tcp://m2m.eclipse.org:1883", TOPIC ) time(0) onFailSwitchTo
      prologFailure;
    println(robot001("waiting for mqttmsg"));
    removeRule tout(X,Y);
    receiveTheMsg m (MSGID,MSGTYPE, SENDER ,robot001,mqttmsg( TOPIC,PAYLOAD )
      , MSGNUM)time(9000);
    [ !? tout(X,Y) ] switchToPlan toutExpired;

    memoCurrentMessage;
    [?? status(X)] removeRule status(X);
    [!? msg( MSGID,TYPE, SENDER ,REC, mqttmsg( TOPIC,PAYLOAD ), MSGNUM ) ]
      println (robot001(value_received(PAYLOAD)));
    [?? msg( MSGID,TYPE, SENDER ,REC, mqttmsg( TOPIC,PAYLOAD ), MSGNUM ) ]
      addRule PAYLOAD;
    [?? status(_, '2')]switchToPlan roomFull;
  }
}

```



```

switchToPlan roomInteraction

Plan roomFull
  [?? room(RFIDNAME,NAME,IP,TOPIC)]println(robot001(room_full_(NAME)));
  solve disconnect time (0) onFailSwitchTo prologFailure;
  println(robot001("searching for another room"));
  switchToPlan search

Plan obstacleavoidance
  println(robot001( "avoiding obstacle" ) );
  println(robot001( "end of the road" ) )

Plan roomInteraction
  println(robot001("interacting with the room"));

  solve actorOp(checkConsistency) time (0) onFailSwitchTo prologFailure;
  [ !? actorOpResult(R)]println( robot001(consistency_check_opresult(R)));
  [!? actorOpResult(false)] switchToPlan alarm;
  [!? room(W,X,Y,Z)] println( robot001(unloading_pallet_in_(X)) );

  [!? room(W,X,Y,Z)] solve actorOp(unloading(X)) time (0)
    onFailSwitchTo prologFailure;

  [!? status(_, '0')] switchToPlan halfinc;
  [!? status(_, '1')] switchToPlan totalinc

Plan halfinc
  [?? status(A,X)] solve publish( "robot001", "tcp://m2m.eclipse.org:1883",
    A , status(A, '1'),2,true)time(0) onFailSwitchTo prologFailure;
  switchToPlan reloading

Plan totalinc
  [?? status(A,X)] solve publish( "robot001", "tcp://m2m.eclipse.org:1883",
    A , status(A, '2'),2,true)time(0)onFailSwitchTo prologFailure;
  switchToPlan reloading

Plan reloading
  println(robot001("Exit from the room"));
  solve disconnect time (0);
  [?? room(RFIDNAME,NAME,IP,TOPIC)] println("garbage collection");
  println(robot001("Loading the next pallet"));
  solve actorOp(loading)time (0);
  println(robot001("Delivering phase"));
  switchToPlan search

Plan alarm
  println(robot001(alarm))

Plan testConnection resumeLastPlan
  solve connect( "robot001", "tcp://m2m.eclipse.org:1883", "unibo/paho/qa")
    time(0) onFailSwitchTo prologFailure;
  solve consult("./pahoTheory.pl") time(0) onFailSwitchTo prologFailure ;
  println(robot001( publish ) ) ;
  solve publish( "robot001", "tcp://m2m.eclipse.org:1883", "unibo/paho/qa",
    "hello1(robot_001,world)")time(0)onFailSwitchTo prologFailure;
  println(robot001( publish ) ) ;
  solve publish( "robot001", "tcp://m2m.eclipse.org:1883", "unibo/paho/qa",
    "hello2(robot_001,world)")time(0)onFailSwitchTo prologFailure;

```

```

        solve disconnect time(0)

    Plan prologFailure
        println( failure(prolog) )

    Plan toutExpired
        [ ?? tout(X,Y) ] println( timeout(X,Y) );
        println("Restarting from the beginning...");
        solve actorOp(unloading) time (0)onFailSwitchTo prologFailure;
        solve actorOp(loading) time (0)onFailSwitchTo prologFailure;
        switchToPlan search
}

```

Room logical behaviour

```

QActor rooma context ctxRoomA -g gray {
    Rules{
        loadTheory(File) :- actorPrintln( loadTheory(File) ),consult( File ).
    }

    Plan init normal
        println(rooma( starts) );
        solve loadTheory("./pahoTheory.pl") time(0) onFailSwitchTo prologFailure;
        addRule rfidName("rfididRoomA");
        addRule roomName("rooma");
        addRule host("localhost");
        addRule port(82);
        addRule info("rfididRoomA","rooma","localhost","rooma");
        addRule iotBrokerAddr("tcp://m2m.eclipse.org:1883");
        addRule status("rooma",'0');

        //Eclipse MQTT Broker tcp://m2m.eclipse.org:1883
        //Mosquitto Broker tcp://localhost:1883
        switchToPlan testConnection;
        switchToPlan loadStatus;
        switchToPlan updateLocalInfo;
        println(rooma( ends) )

    Plan testConnection resumeLastPlan

        solve connect("rooma", "tcp://m2m.eclipse.org:1883", "rooma")
            time(0) onFailSwitchTo prologFailure;
        println(rooma(saving_state));
        [!? status(A,X)] solve publish( "rooma", "tcp://m2m.eclipse.org:1883",
            "rooma", status(A,X) ,2,true)time(0)onFailSwitchTo prologFailure

    Plan loadStatus resumeLastPlan

        println(rooma("asking for previous saved state"));
        solve subscribe("rooma", "tcp://m2m.eclipse.org:1883", "rooma")
            time(0) onFailSwitchTo prologFailure;
        receiveTheMsg m (MSGID,MSGTIPE,rooma,rooma,MSGCONTENT, MSGNUM)time(1000);
        printCurrentMessage;
        memoCurrentMessage;

        [?? status(_,X)] removeRule status(A,_,X);
        [?? msg( MSGID,TYPE, SENDER ,REC, mqttmsg( TOPIC,PAYLOAD ), MSGNUM ) ]
            addRule PAYLOAD;

```

```

        [!? status(_,X)]println(rooma(status(X)))

Plan updatelocalinfo
    println(rooma( "waiting for any change" ) );
    receiveTheMsg m (MSGID,MSGTIPE,SENDER,RECEIVER,mqttmsg( TOPIC,PAYLOAD),
        MSGNUM)time(7000);
    memoCurrentMessage;
    [?? status(X)] removeRule status(X);
    [!? msg( MSGID,TYPE, SENDER ,REC, mqttmsg( TOPIC,PAYLOAD ), MSGNUM ) ]
        println (rooma(value_received(PAYLOAD)));
    [?? msg( MSGID,TYPE, SENDER ,REC, mqttmsg( TOPIC,PAYLOAD ), MSGNUM ) ]
        addRule PAYLOAD;
    repeatPlan

Plan rfidSig
    println(rooma( "emmitting rfid signal" ) );
    receiveMsg time (3000) react event palletunloaded -> accountingPallet ;
    [!? info(RFIDNAME,NAME,IP,TOPIC)]onMsg getRoomInfo : X -> replyToCaller
        -m roomInfo : info(RFIDNAME,NAME,IP,TOPIC)
    repeatPlan

Plan toutExpired
    [ ?? tout(X,Y) ] println( timeout(X,Y) )

    Plan prologFailure
        println( failure(prolog) )
}

```

6.3.7 Workflow for QActors - Phase 6

Taking another step between analysis and design, we specify the logical architecture of the system in java and we build the first prototype by selecting a working environment.

```

----- Robot's Context -----
package it.unibo.ctxRobot;
import it.unibo.qactors.ActorContext;
import java.io.InputStream;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.system.SituatedSysKb;
public class MainCtxRobot extends ActorContext{
//private IBasicEnvAwt env;

    public MainCtxRobot(String name, IOutputEnvView outEnvView,
        InputStream sysKbStream, InputStream sysRulesStream) throws Exception {
        super(name, outEnvView, sysKbStream, sysRulesStream);
        this.outEnvView = outEnvView;
        env = outEnvView.getEnv();
    }
    @Override
    public void configure() {

```

```

        try {
            SitedSysKb.init();          //Init the schedulers
            println("Starting the actors .... ");
            new it.unibo.robot001.Robot001("robot001", this, outEnvView);

            } catch (Exception e) {
                e.printStackTrace();
            }
        }

/*
* -----
* MAIN
* -----
*/

public static void main(String[] args) throws Exception{
    IOutputEnvView outEnvView = SitedSysKb.standardOutEnvView;
    InputStream sysKbStream =
        new java.io.FileInputStream("./srcMore/it/unibo/ctxRobot/modeliot.pl");
    InputStream sysRulesStream=MainCtxRobot.class.getResourceAsStream("sysRules.pl");
    new MainCtxRobot("ctxRobot",outEnvView,sysKbStream,sysRulesStream ).configure();
}
}

```

Room's Context

```

package it.unibo.ctxRoomA;
import it.unibo.qactors.ActorContext;
import java.io.InputStream;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.system.SitedSysKb;
public class MainCtxRoomA extends ActorContext{
//private IBasicEnvAwt env;

    public MainCtxRoomA(String name, IOutputEnvView outEnvView,
        InputStream sysKbStream, InputStream sysRulesStream) throws Exception {
        super(name, outEnvView, sysKbStream, sysRulesStream);
        this.outEnvView = outEnvView;
        env = outEnvView.getEnv();
    }
    @Override
    public void configure() {
        try {
            SitedSysKb.init();          //Init the schedulers
            println("Starting the actors .... ");
            new it.unibo.rooma.Rooma("rooma", this, outEnvView);

            } catch (Exception e) {
                e.printStackTrace();
            }
        }

/*
* -----
* MAIN
* -----
*/

```

```

*/
public static void main(String[] args) throws Exception{
    IOutputEnvView outEnvView = SituatedSysKb.standardOutEnvView;
        InputStream sysKbStream    =
            new java.io.FileInputStream("./srcMore/it/unibo/ctxRoomA/modeliot.pl");
    InputStream sysRulesStream=MainCtxRoomA.class.getResourceAsStream("sysRules.pl");
    new MainCtxRoomA("ctxRoomA",outEnvView,sysKbStream,sysRulesStream ).configure();
    }
}

```

Robot's Class

```

package it.unibo.robot001;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.ActorContext;

public class Robot001 extends AbstractRobot001 {
    int counter = 0;
    //int pallet = 1;
    public Robot001(String actorId, ActorContext myCtx, IOutputEnvView outEnvView )
        throws Exception{

        super(actorId, myCtx, outEnvView);
    }
    public boolean checkconsistency(){
        return true;
    }

    }
    public void unloading(String room){
        this.removeRule("pallet");

        //this.platform.raiseEvent(room, "palletunloaded", "X");
    }
    public void loading(){
        //pallet++;
        counter = 0;
        this.addRule("pallet");
    }

    }

    public void scanningforrooms(){
        counter++;
        switch(counter){
            //iniezione tupla contenente nome della stanza es:
            //room(RFIDNAME,NAME,IP,TOPIC)
            case 1:
                this.addRule("room(rfididRoomA,rooma,localhost,rooma)");
                break;
            case 2:
                this.addRule("room(rfididRoomB,roomb,localhost,roomb)");
                break;
            case 3:
                this.addRule("room(rfididRoomC,roomc,localhost,roomc)");
                break;
            case 4:

```

```

        counter = 0;
        this.platform.raiseEvent("robot001", "obstacledetected", "X");

        default :
            break;

    }
    try {

        sleep(2000);

        this.platform.raiseEvent("robot001", "roomdetected", "X");

    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
public void endevent(){
    //this.platform.unregisterForEvent( "roomdetected");
    this.platform.unregisterForAllEvents("robot001");
}
}

```

Room's Class

```

package it.unibo.rooma;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.ActorContext;

public class Rooma extends AbstractRooma {
    public Rooma(String actorId, ActorContext myCtx, IOutputEnvView outEnvView )
        throws Exception{
        super(actorId, myCtx, outEnvView);
    }
}

```

MQTT Utils

```

package it.unibo.paho.utils;
import it.unibo.contactEvent.interfaces.IContactEventPlatform;
import it.unibo.contactEvent.platform.ContactEventPlatform;
import it.unibo.qactors.QActor;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;

public class MqttUtils implements MqttCallback{

    private static String topic = "it/unibo/paho/iot/rooma";
}

```

```

private static String broker = "tcp://m2m.eclipse.org:1883";
//private static String broker = "tcp://localhost:1883";
private String broker = "tcp://m2m.eclipse.org:1883";
private static String clientId1 = "qa_unibo_1";
private static String clientId2 = "qa_unibo_2";

private static MqttUtils myself = null;

protected IContactEventPlatform platform ;
protected String clientId = null;
protected String eventId = "mqtt";
protected String eventMsg = "";
protected QActor actor = null;
protected MqttClient client = null;

public MqttUtils(){
    try {
        platform = ContactEventPlatform.getPlatform();
        myself = this;
    } catch (Exception e) {
        e.printStackTrace();
    }
}

protected void doTest() throws Exception{
    publish(null,clientId1,broker,topic,"hello(world)",1,true);
    subscribe(null,clientId2,broker,topic);
}

public static MqttUtils getMqttSupport( ){
    return myself ;
}

public void connect(QActor actor, String brokerAddr, String topic )
    throws MqttException{
    clientId = MqttClient.generateClientId();
    connect(actor, clientId, brokerAddr, topic);
}

public void connect(QActor actor,String clientId,String brokerAddr,String topic)
    throws MqttException{
    System.out.println("connect "+ clientId );
    this.actor = actor;
    client = new MqttClient(brokerAddr, clientId);
    MqttConnectOptions options = new MqttConnectOptions();
    options.setWill("unibo/clienterrors", "crashed".getBytes(), 2, true);
    client.connect(options);
}

public void disconnect( ) throws MqttException{
    System.out.println("disconnect "+ client );
    if( client != null ) client.disconnect();
}

public void publish(QActor actor, String clientId, String brokerAddr,
    String topic, String msg, int qos, boolean retain) throws MqttException{
    MqttMessage message = new MqttMessage();
    message.setRetained(retain);
    if( qos == 0 || qos == 1 || qos == 2){
        //qos=0 fire and forget; qos=1 at least once (default);
        //qos=2 exactly once
        message.setQos(0);
    }
}

```

```

        }
        message.setPayload(msg.getBytes());
        client.publish(topic, message);
        System.out.println("publish done by "+ clientid );
    }

    public void subscribe(QActor actor, String clientid, String brokerAddr,
        String topic) throws Exception {
        try{
            System.out.println("subscribe "+ clientid + " on " + topic);
            this.actor = actor;
            //      MqttClient client = new MqttClient( brokerAddr, clientid);
            client.setCallback(this);
            client.subscribe(topic);
        }catch(Exception e){
            System.out.println("subscribe error "+ e.getMessage() );
            //      eventMsg = "mqtt("+topic +",\""+e.getMessage()+"\"";
            eventMsg = "mqtt(" + eventId +", failure)";
            System.out.println("subscribe error "+ eventMsg );
            //sense in qa has not been yet executed: the event is
            //lost platform.raiseEvent("mqttutil",eventId,eventMsg );
            if( actor != null ) actor.sendMessage("mqttmsg",
                actor.getName(), "dispatch", "error");
            throw e;
        }
    }

    @Override
    public void connectionLost(Throwable cause) {
        System.out.println("connectionLost = "+ cause.getMessage() );
    }

    @Override
    public void deliveryComplete(IMqttDeliveryToken token) {
        System.out.println("deliveryComplete token= "+ token );
    }

    @Override
    public void messageArrived(String topic, MqttMessage msg) throws Exception {
        System.out.println("messageArrived on "+ topic + "="+msg.toString());
        String mqttmsg = "mqttmsg(" + topic +", " + msg.toString() +)";
        System.out.println("messageArrived mqttmsg "+ mqttmsg);
        //      platform.raiseEvent("mqttutil", eventId, mqttmsg );
        if( actor != null ) actor.sendMessage("mqttmsg", actor.getName(),
            "dispatch", mqttmsg);
    }

    public static void main(String[] args) throws Exception {
        new MqttUtils().doTest();
    }
}

```

MQTT operations have been implemented according to the prolog syntax.

Paho Theory

```

/*
=====
pahoTheory.pl
=====
*/
connect( Name, BrokerAddr, Topic ):-
    java_object("it.unibo.paho.utils.MqttUtils", [], UMQTT),
    actorobj(A),
    actorPrintln( connect( UMQTT , A ) ),
    UMQTT <- connect(A, Name, BrokerAddr, Topic ).
disconnect :-
    actorPrintln( disconnect ),
    class("it.unibo.paho.utils.MqttUtils") <- getMqttSupport returns UMQTT,
    actorPrintln( disconnect( UMQTT ) ),
    UMQTT <- disconnect.

publish( Name, BrokerAddr, Topic, Msg, Qos, Retain ):-
    actorPrintln( publish( BrokerAddr, Topic, Msg, Qos, Retain ) ),
    actorobj(A),
    %% java_object("it.unibo.paho.utils.MqttUtils", [], UMQTT),
    class("it.unibo.paho.utils.MqttUtils") <- getMqttSupport returns UMQTT,
    actorPrintln( publish( UMQTT ) ),
    UMQTT <- publish(A, Name, BrokerAddr, Topic, Msg, Qos, Retain).

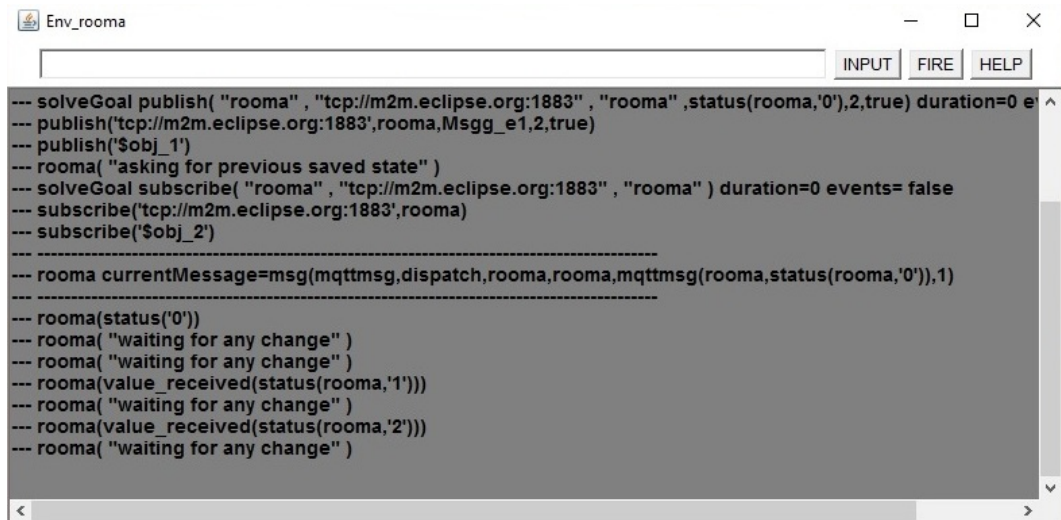
subscribe( Name, BrokerAddr, Topic ):-
    actorPrintln( subscribe( BrokerAddr, Topic ) ),
    actorobj(A),
    %% java_object("it.unibo.paho.utils.MqttUtils", [], UMQTT),
    class("it.unibo.paho.utils.MqttUtils") <- getMqttSupport returns UMQTT,
    actorPrintln( subscribe( UMQTT ) ),
    UMQTT <- subscribe(A, Name, BrokerAddr, Topic ).

/*
-----
initialize
-----
*/
initialize :-
    actorPrintln("pahoTheory started ...") .

:- initialization(initialize).

```

6.4 The system running

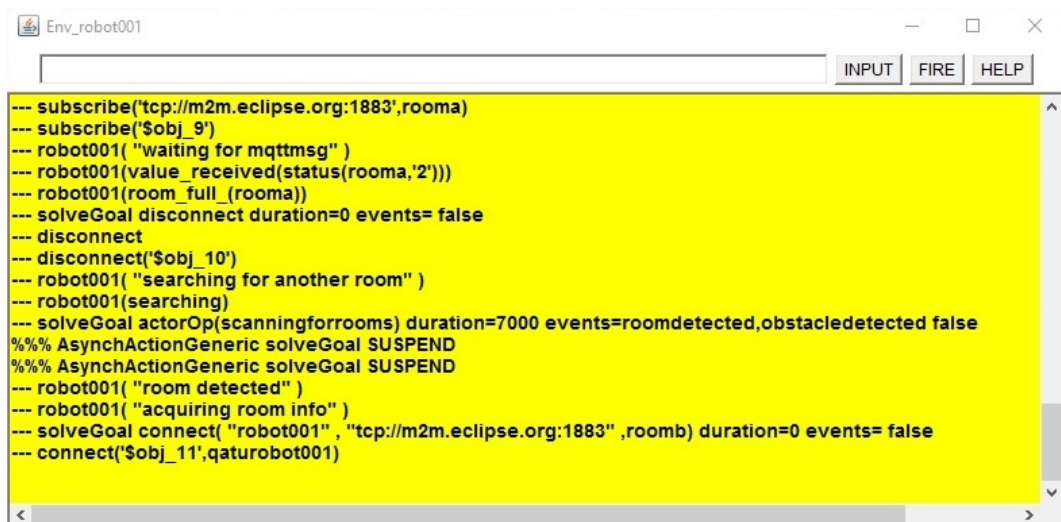


```

--- solveGoal publish( "rooma" , "tcp://m2m.eclipse.org:1883" , "rooma" ,status(rooma,'0'),2,true) duration=0 e
--- publish("tcp://m2m.eclipse.org:1883',rooma,Msgg_e1,2,true)
--- publish("$obj_1")
--- rooma( "asking for previous saved state" )
--- solveGoal subscribe( "rooma" , "tcp://m2m.eclipse.org:1883" , "rooma" ) duration=0 events= false
--- subscribe("tcp://m2m.eclipse.org:1883',rooma)
--- subscribe("$obj_2")
-----
--- rooma currentMessage=msg(mqttmsg,dispatch,rooma,rooma,mqttmsg(rooma,status(rooma,'0')),1)
-----
--- rooma(status('0'))
--- rooma( "waiting for any change" )
--- rooma( "waiting for any change" )
--- rooma(value_received(status(rooma,'1')))
--- rooma( "waiting for any change" )
--- rooma(value_received(status(rooma,'2')))
--- rooma( "waiting for any change" )

```

Figure 6.3: Room's log.



```

--- subscribe("tcp://m2m.eclipse.org:1883',rooma)
--- subscribe("$obj_9")
--- robot001( "waiting for mqttmsg" )
--- robot001(value_received(status(rooma,'2')))
--- robot001(room_full_(rooma))
--- solveGoal disconnect duration=0 events= false
--- disconnect
--- disconnect("$obj_10")
--- robot001( "searching for another room" )
--- robot001(searching)
--- solveGoal actorOp(scanningforrooms) duration=7000 events=roomdetected,obstacle detected false
%%% AsynchActionGeneric solveGoal SUSPEND
%%% AsynchActionGeneric solveGoal SUSPEND
--- robot001( "room detected" )
--- robot001( "acquiring room info" )
--- solveGoal connect( "robot001" , "tcp://m2m.eclipse.org:1883" ,roomb) duration=0 events= false
--- connect("$obj_11',qaturobot001)

```

Figure 6.4: Robot's log.

Chapter 7

Conclusions

The proliferation of entities with sensing capabilities is bringing closer the vision of an Internet of Things which is expected to become mainstream in the next 5-10 years. Thanks to IoT new capabilities are made possible through the access of rich new information sources. The experiment undertaken in chapter 6 is only one simple example among the countless that shows us the way in which IoT will enter into our lives in the near future.

To support lots of heterogeneous and distributed entities new paradigms have been recognized as the *Fog Computing* that arises as a structure close to the data sources offering geolocation services.

Because of the expectations raised, different providers like Microsoft, Netflix, Amazon and IBM, have begun the design, development and deployment of Cloud solutions to optimize the utilization of their data centers. Some open-source solutions are being consolidated and among them is becoming pervasive the use of OpenStack and Cloud Foundry. This place between IaaS and PaaS level has been identified as the space for DevOps philosophy to flourish allowing agile changes and continuous integration by small teams which are responsible for the entire lifecycle of a microservice.

Taking meta concepts, emerging from the Internet of Things and Cloud environments, as microservices, and injecting them in a Model Driven software factory enabled us to obtain an automatic code generation.

This set of foundational technologies built on top of each other has enabled us with new ways of building and running technology. Thanks to the Model Driven approach the lost top-down philosophy has been regained.

In conclusion, not every question arisen has been closed. As a future direction we will have to tackle the mapping issue between the abstractions supported by today languages and the microservices. Should agents and actors integrate microservices or contrarily should microservices embody agents and actors? The abstraction gap identified is certainly not null. The answer lies somewhere and it will be strictly related to the formalization process of microservices in the next future.

Bibliography

- [1] Amazon simple storage service (amazon s3). Retrieved from <http://aws.amazon.com/s3/>, pages 1–6, 2011.
- [2] Cloud foundry. Retrieved from <http://docs.cloudfoundry.org/>, 2016.
- [3] Eduardo A Patrocinio Henryk Gorski Manav Gupta Patrick M Ryan Richard Osowski Ryan C Livesey Vasfi Gucer Ann Marie Fred, Bhargav Perepa. *IBM Bluemix Architecture Series: Web Application Hosting on IBM Containers Leveraging best practice and reference architectures for cloud*. IBM Redbooks, 2015.
- [4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.
- [5] K. Ashton. That “internet of things” thing. *RFID Journal*, 4986:17–86, 2009.
- [6] A. Banks and R. Gupta. Mqtt version 3.1.1. *OASIS Standard*. <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>.
- [7] S. Bhardwaj, L. Jain, and S. Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of Engineering and Information Technology*, 2(21):60–63, 2010.

- [8] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.
- [9] R. Caceres and A. Friday. *Ubicomp systems at 20: Progress, opportunities, and challenges*. IEEE Pervasive Computing, 2012.
- [10] Lucas Carlson. *Programming for PaaS*. O’Reilly, 2013.
- [11] S. Chen, H. Xu, D. Liu, B. Hu, and H. Wang. *A vision of IoT: Applications, challenges, and opportunities with China Perspective*. IEEE Internet of Things Journal. Institute of Electrical and Electronics Engineers Inc, 2014.
- [12] C. A. Cois, J. Yankel, and A. Connell. Modern devops: Optimizing software development through effective system interactions. *IEEE International Professional Communication Conference*, 2015.
- [13] A. Corradi, M. Fanelli, and L. Foschini. Vm consolidation: A real case based on openstack cloud. *Future Generation Computer Systems*, 32(1):118–127, 2014.
- [14] M. Eysholdt and H. Behrens.
- [15] Christopher Ferris and Joel Farrell. What are web services? *Commun. ACM*, 46(6):31–, June 2003.
- [16] R. T. Fielding. Architectural styles and the design of network-based software architectures. *Building*, 54:162, 2000.
- [17] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. *Grid Computing Environments Workshop*, 2008:1–10, 2008.
- [18] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future direc-

- tions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.
- [19] Liming Zhu Len Bass, Ingo Weber. *DevOps A Software Architect's Perspective*. Addison-Wesley, 2015.
- [20] Ang Li, Xiaowei Yan, Srikanth Kandula, and Ming Zhang. Cloud-cmp: Comparing public cloud providers. pages 1–14, 2010.
- [21] A. Natali and A. Molesini. *La costruzione dei sistemi software: dai modelli al codice*. Progetto Leonardo, 2008.
- [22] Sam Newman. *Building Microservices designing fine-grained systems*. O'Reilly, 2015.
- [23] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs "big" web services: Making the right architectural decision categories and subject descriptors. *Technology*, pages 805–814, 2008.
- [24] Dan Sanderson. *Programming Google App Engine*. O'Reilly, 2009.
- [25] I. Stojmenovic and S. Wen. The fog computing paradigm: Scenarios and security issues. 2:1–8, 2014.
- [26] J. Thones. *Microservices*. 2015.
- [27] J. Varia and S. Mathew. Overview of amazon web services. *White Paper*, pages 1–18, 2012.
- [28] M. Weiser. *The Computer for the 21st Century*. Scientific American, 1991.