POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

POLITECNICO
MILANO 1863

Liquid Android: A Middleware for managing Android Intents in a Distributed Net over Wi.Fi

Relatore:

Prof. Luciano Baresi

Tesi di Laurea di:

Marco Molinaroli

Matricola:

837721

Anno accademico 2015–2016

Ringraziamenti

Ringraziamenti.

Milano, April 2017 Marco

Sommario

Parole chiave: PoliMi, Tesi, LaTeX, Scribd

Abstract

Text of the abstract in english...

Keywords: PoliMi, Master Thesis, LaTeX, Scribd

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Nowadays Android is the most common mobile operating system (OS), and it is now clear which it is not only a tiny operating system, but a full functional OS to be used for general purpose. One of the most peculiar characteristic of the Android OS is which it can be installed in a variety of devices such as *"handled"*, like smart-phones and tablets, *"wearable"*, like smart-watches, but also in other kind of things like standard desktops and laptops, smart-tv and tv boxes, and so on.
The great variety of devices described above can run and benefit all the functions of the Android OS which is acknowledged for its ease of use, and the great abundance of applications, with which users can do almost everything.
However one of the greatest limits of Android is that the system was designed to run on the top of a virtual machine and each application which can be executed starts a Linux process which has its own virtual machine (VM), so an application's code runs in isolation from other apps. This technique is called *"app sandboxing"* and it is used to guarantee an high level of security, because different applications can not read write, or worse steal, data and sensible information from other applications.That is, each app, by default, has access only to the components that it requires to do its work and no more. This creates a very secure environment in which an app cannot access parts of the system for which it is not given permission.
Under this limitations the Android OS provides a mechanism to make communicate the various component of the applications and the operating system itself : the so called *"intents"*. An intent is an abstract description of an operation to be performed,it provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities. However, even do the intents can be created and resolved within the same android running devices, there is not a mechanism that can send and resolve intents from a devices to another one.
In a world where computers are everywhere and can do almost everything and can communicate among them in different but efficient ways, the fact that android devices are not able to easily exchange intents is such a major limitation to

the android users. As we know our world is fast moving to a world of *"ubiquitous computing"* where there is no more a single *"fat calculator"* but a variety of multipurpose and specialized devices. In this world of pervasive computation, Android devices are widespread, cheap and powerful enough to do most of the things that we can imagine and would be great if they can be used together in a smart way. The aim of this thesis work is to study enough the android framework to find a solution to this problem, and create a middleware to extend the Android OS, creating a distributed system in which intents can be generated from one device and resolved by others in a net connected in a LAN. This can help developers build distributed native Android application to exploit the power of any different device running the OS and let the users use their own devices such as they were one single big device.
Each sentence or technology, that may appear not clearly explained here for the reader, is further discussed and clarified in next chapters.

1.2 Outline

The thesis is organized as follows:

In the second chapter the state of art is described: a full overview on current technologies, ideas and issues is provided.

In the third chapter I have defined the faced problem, its constraints and its boundaries.

In the fourth chapter

In the fifth chapter

In the sixth chapter

Chapter 2

State of the Art

2.1 Android OS

As already mentioned in 1.1, the Android operating system is an open source OS developed by Google based on Linux kernel, that can be installed on many different kind of devices.
In this section i want to give to the reader the basic knowledge of the Android framework to understand why and how the operating system works.

2.1.1 Brief History

The Android era officially began on October 22nd, 2008, when the *T-Mobile G1* launched in the United States [**verge2011android**].
At that moment the company of mountain view, Google, felt the need to create a new operating system which was able to be installed on most modern mobile phones of the time. To meet this need the Google engineers created an OS that was based on the Linux kernel, lightweight enough and ease to be used with simple hand gestures by touching the screen of the phone.

Figure 2.1: The T-Mobile G1 and the Android 1.0 menù

The main characteristic of the OS were and are also now:

- The pull-down notification window.
- Home screen widgets.
- The Android Market.
- Google services integration (eg. Gmail).
- Wireless connection technologies (eg Wi-Fi and Bluetooth)

The success of the first version of the brand new mobile operating system and the open source philosophy guaranteed the fast spread of the Android devices all over the world. In few years Google improved and released many version of the OS and with the help of the market growth Android has become a complete os. In the table below there is a brief description of the various distribution of the Android OS at the time of writing of this document.

As we can see in Table 2.1 there are, currently, 25 level of the Android *API*

Table 2.1: Android versions

Name	Version	Release Date	API Level
Alpha	1.0	September 23, 2008	1
Beta	1.1	February 9, 2009	2
Cupcake	1.5	April 27, 2009	3
Donut	1.6	September 15, 2009	4
Eclair	2.0 – 2.1	October 26, 2009	5–7
Froyo	2.2 – 2.2.3	May 20, 2010	8
Gingerbread	2.3 – 2.3.7	December 6, 2010	9–10
Honeycomb	3.0 – 3.2.6	February 22, 2011	11–13
Ice Cream Sandwich	4.0 – 4.0.4	October 18, 2011	14–15
Jelly Bean	4.1 – 4.3.1	July 9, 2012	16–18
KitKat	4.4 – 4.4.4	October 31, 2013	19
Lollipop	5.0 – 5.1.1	November 12, 2014	21–22
Marshmallow	6.0 – 6.0.1	October 5, 2015	23
Nougat	7.0 – 7.1.1	August 22, 2016	24–25

(Application programming interface) which developers can use to build Android applications. In particular various API levels introduce innovations in the OS but, applications developed using an higher *API level* can not be executed in a device running lower versions of the operating system. This is a second major limitations for the *"Android ecosystem"*, moreover as mentioned before, the Android OS is released under an open source license, which is great for the developer, but which prevents Google to provide updates, in a centralized way, to all devices. For this reason there are currently many active devices running different versions of the mobile OS, as we can check in Table 2.2, which shows, in percentage, the fragmentations of active machines running Android OS.

Table 2.2: Android OS versions fragmentation

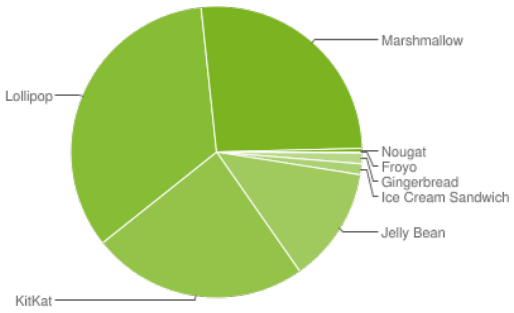


Figure 2.2: Android OS fragmentation chart

Version	API Level	Distribution
2.2	8	0.1%
2.3.3 - 2.3.7	10	1.2%
4.0.3 - 4.0.4	15	1.2%
4.1.x	16	4.5%
4.2.x	17	6.4%
4.3	18	1.9%
4.4	19	24.0%
5.0	21	10.8%
5.1	22	23.2%
6.0	23	26.3%
7.0	24	0.4%

Data in Table 2.2 were collected during a 7-day period ending on December 5, 2016, by Google. Any versions with less than 0.1% distribution are not shown **[devandroiddash**].

2.1.2 Structure

Android is an operating system based on the Linux kernel. The project responsible for developing the Android system is called the *Android Open Source Project (AOSP)* and it lead by Google.

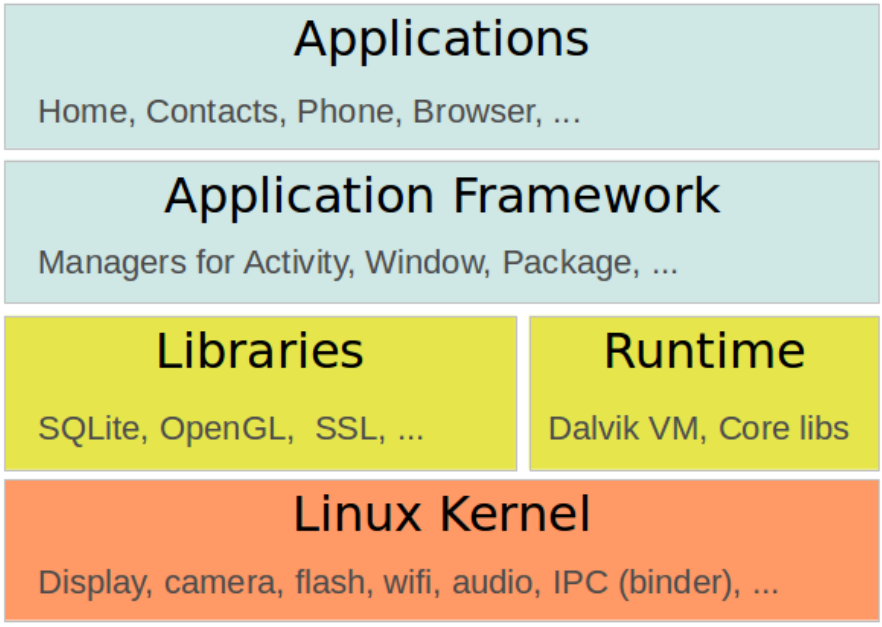


Figure 2.3: Android OS 4 layers

The OS can be divided into the four layers as depicted the Figure 2.3. An Android application developer typically works with the two layers on top to

create new Android applications [**vogel2016android**].

Linux Kernel is the most flexible operating system that has ever been created. It can be tuned for a wide range of different systems, running on everything from a radio-controlled model helicopter, to a cell phone, to the majority of the largest supercomputers in the world [**hartman2006linux**]. This is in practice the communication layer for the underlying hardware.

Runtime and Libraries Runtime is the term used in computer science to designate the software that provides the services necessary for the execution of a program.There are two different *"runtime systems"* which can work with the Android OS:

- *Dalvik VM* is an optimized version for low memory devices of the *Java Virtual Machine (JVM)* used in Android 4.4 and earlier version. It is stack based and it works by converting using a *just-in-time (JIT)*, each time an application is executed, Android's *bytecode* into machine code.
- *ART (Android Runtime)* introduced with Android 4.4 KitKat. This runtime uses an *AOT (Ahead-of-Time)* approach, with which code is compiled during the installation of an application and then is ready to be executed.

Standard Android libraries are for many common framework functions, like, graphic rendering, data storage, web browsing. [**vogel2016android**]. This layer contains also standard *java libraries*.

Application Framework is the layer that contains the Android components for the application such as activities, fragments, services and so on.

Applications are pieces of software written in *java code* running on top the other layers.

2.1.3 Application Framework

In this section I want to give some details of the application composition and work flow to better understand the subsequent sections in which I will describe the given problem and the proposed solution.
As briefly described in 2.1.2 the Android application framework *("AppFramework")* is the core of the Android *development API*. It contains useful and needed components to build native apps.
The main components with which each application is composed are:

Intents are objects that initiate actions from other app components, either within the same program *(explicit intents)* or through another piece of software on the device *(implicit intents)*. Acconrding to the official Google's Android for developer documentation, an Intent is a sort of messaging object which can be

used to request an action from another application component (eg. activities). There are three fundamental use cases:

- Starting an activity: we will see that activities represents a single screen in Android applications, intents allow to start activities by describing them and carrying any necessary data.
- Starting a service: I will explain later in deeper details that services are component which performs operations in background. As for the activities, services are initialized through intent and in the same way they describe the service to start and carries any necessary data.
- Delivering a broadcast: broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging.

As already mentioned there are mainly two categories of intents:

- explicit intents, used when it is needed to start component within the same application. As the name implies explicit intents call components by using by name (the full *class object* name), for example, it is possible to start a new activity in response to a user action or start a service to download a file in the background.
- implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map [**devandroidintent**].

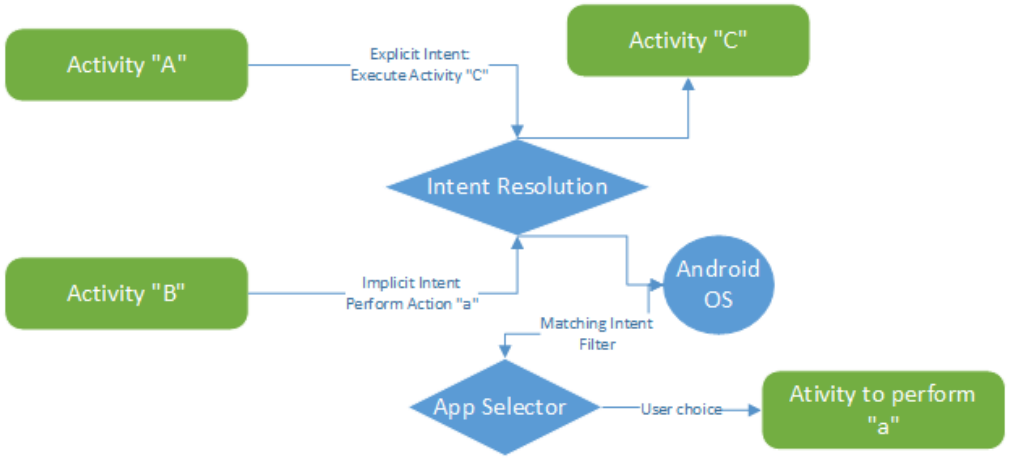


Figure 2.4: Intent resolution mechanism

The Figure 2.4 explains well how an intent is resolved by the OS whether it is implicit or explicit. When an implicit intent needs to be resolved, the OS searches applications which can handle it by means of *intent filters*.A Intent filter specifies

the types of intents that an activity, service, or broadcast receiver can respond to. The Android System searches all apps for an intent filter that matches the intent to be resolved. When a match is found, the system starts the matching component, or, if there are more than one, let the user select the preferred action to be performed.

Activities are one of the fundamental building blocks of apps on the Android platform. They serve as the entry point for a user's interaction with an app, and are also central to how a user navigates within an app. [**devandroidactivity**]. An activity is the entry point for interacting with the user. It represents a single screen with a user interface *GUI*: in this way activities are containers for other Android's GUI elements (eg. buttons, textviews,...).

Services is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface [**devandroifundamentals**].

Broadcast Receivers are components that enable the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to applications that aren't currently running [**devandroifundamentals**].

2.1.4 Security

As described in 2.1.1 Android was born to be a good mobile OS and it is mainly for this reason that the system is designed to protect personal and sensible data form malicious guys.
Like the rest of the system, Android's security model also takes advantages of the security features offered by the Linux kernel. Linux is a *multiuser OS* and its kernel can isolate user data from one another: one user can not access another user's file unless explicitly granted permission. Android takes advantages of this user isolation, considering each application a different user provided with a dedicated *UID (User ID)* [**elenkov2014android**] Android in fact, is designed for smartphones that are personal devices and do not need, usually, a multi physical user support. The most important security techniques adopted by Android are:

Application Sandboxing Android automatically assigns a unique *AppID* (Linux UID) when an application is installed and then executed that specific app in a dedicated process as that UID. This technique isolate all the applications at process level and additionally each app has permissions to read/write a specific and dedicated directory.

Permissions Since application are sandboxed and do not have the rights to read/write date outside them, it is possible to grant additional rights to android applications by explicitly asking them. Those access rights are called *permission*. Applications can request permissions by listing them in a configuration file called *android manifest*. In Android 5.1 and earlier versions permission are inspected and granted at installation time, when the user is alerted with a dialog box in which are listed permissions the application to be installed needs to work properly and when granted cannot be revoked. Starting from android 6.0 permission are asked the first time that an application need them, and when are granted they can be revoked manually in the OS settings for that specific application.

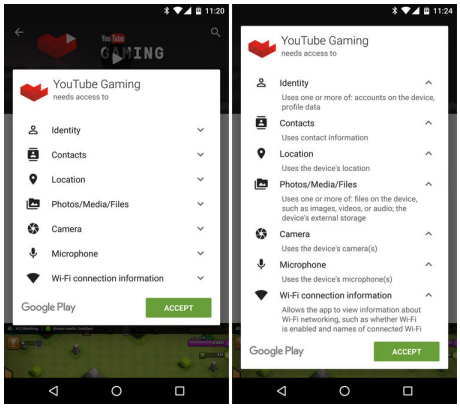


Figure 2.5: Android 5.1- permission example

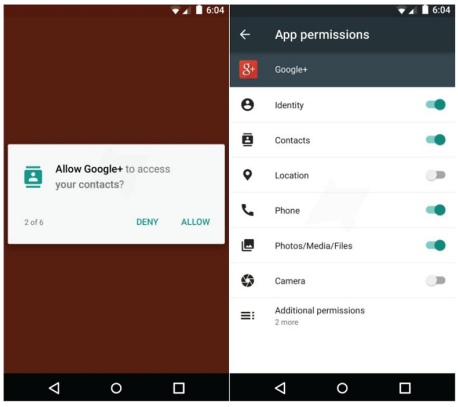


Figure 2.6: Android 6.0+ permission example

Figure 2.7: Android permission Examples

SeLinux Security Enhanced Linux, is a *mandatory access control (MAC)* system for the Linux operating system. With a MAC the operating system constrains the ability of a subject or initiator to access or generally perform some sort of operation on an object or target. Starting in Android 4.3, SELinux provides a mandatory access control (MAC) umbrella over traditional discretionary *access control (DAC)* environments. For instance, software must typically run as the root user account to write to raw block devices. In a traditional DAC-based Linux environment, if the root user becomes compromised that user can write to every raw block device. However, SELinux can be used to label these devices so the process assigned the root privilege can write to only those specified in the associated policy. In this way, the process cannot overwrite data and system settings outside of the specific raw block device [**secure2017android**].

2.1.5 Connectivity

As already amply explained previously many Android design choices are due to the fact that it was thought for mobile devices which must have connectivity to intercommunicate among them.
With the evolution of various wireless communication technologies, Android

devices, nowadays, are equipped whit different kinds of modulus, the most common are:

- Wi-Fi
- Bluetooth
- NFC
- Cellular Network

The Android Os provide a full library to operate with these technologies and it is possible to integrate in applications the possibility to communicate over these wireless modules. With the *Android connectivity API* data can be send and received in an efficient way.

I have only quickly listed some features and possible issues of my source, to have a complete idea it is possible to read all the official Android documentation in [**devandroifundamentals**].

2.2 Distributed System

In this section I want to give to the reader some basics about distributed systems, including technical details and examples to make the proposed solution easier to understand.

2.2.1 Definition

A distributed system is a collection of independent computers that appears to its users as a single coherent system.
This definition has several important aspects. The first one is that a distributed system consists of components (i.e., computers) that are autonomous. A second aspect is that users (be they people or programs) think they are dealing with a single system. This means that one way or the other the autonomous components need to collaborate [**tanenbaum2010distributed**].
In Figure 2.8 it is possible to see how can be structured a distributed system: a the top we have the real distributed application, which is the final interface to be used, under which it is possible to have different combinations of services used to make communicate different machines that may use different operating systems. The real magic is done by the layer called *middleware service* in the picture. A middleware in computer science is a set of software which act as intermediaries between structures and computer programs, allowing them to communicate in spite of the diversity of protocols or running OSs.

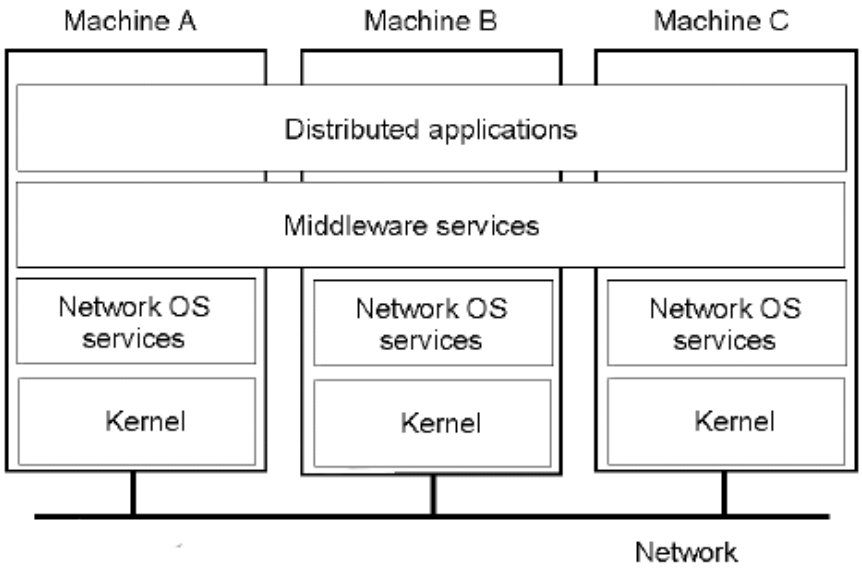


Figure 2.8: Distributed system structure

2.2.2 Challenges

There are many challenges in distributed systems field: distributed applications are often really complex and easily exposed to physical and technical failures because of their nature. Major challenges and property to be considered when developing a system of this kind are:

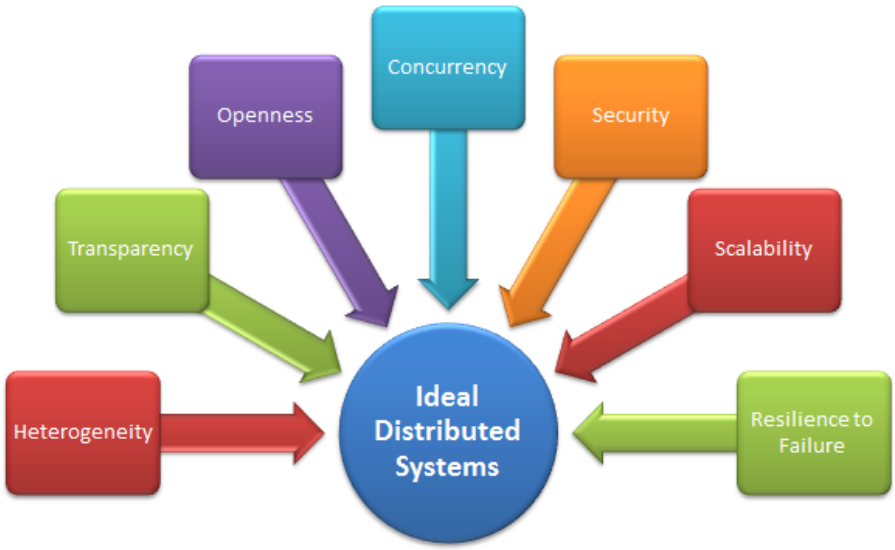


Figure 2.9: Distributed system challenges

- Heterogeneity, is a major challenge because there are many different component to be considered, distributed systems may be developed for example for different hardware, networks, operating systems and programming languages.

- Openness, determines whether a system can be extended and reimplemented in various ways, so distributed systems should use standards as much as possible. Developers should always choose the simplest ways during design and implementation phases.
- Security, is crucial in many areas of computer science and specially in distributed systems, where data are exchanged by a several number of machines.
- Scalability, is the ability to easily increase the size of the system in terms of users/resources and geographic span.
- Failure handling, is important because having different components working together to a common goal means that distributed system can fail in many ways. This raises some issue: it would be nice id distributed systems can detect, mask and tolerate failures.
- Concurrency in distributed systems is a matter of fact, access to shared resources (information or services) must be carefully synchronized.
- Transparency level are listed in Table 2.3

Table 2.3: Transparency levels

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource may be shared by several competitive users
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

2.2.3 Comunication Model

Remote procedure call (RPC)

Remote method invocation (RMI)

Message oriented

2.2.4 Architectures

There are actually many different kinds of distributed systems which can be classified in by means of their architecture composition.

Client-Server is the most common architecture in computer systems, there are many variants depending on the internal division of its components but it has a common separation of duties. Server side components are passive and wait for clients invocations.Client computers provide an interface to allow a computer user to request services of the server and to display the results it returns. Servers wait for requests to arrive from clients and then respond to them. Ideally, a server provides a standardized transparent interface to clients so that clients need not be aware of the specifics of the system (i.e., the hardware and software) that is providing the service. The communication adopted by these kind of systems is message oriented or through RPC.

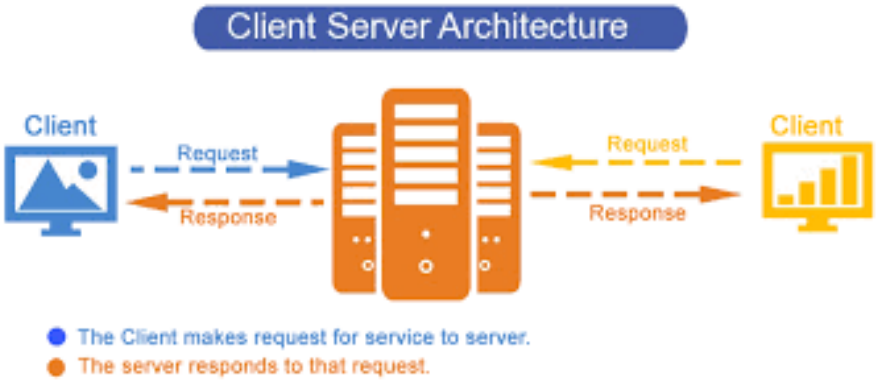


Figure 2.10: Client server architecture

Peer-to-Peer (P2P) is a fully distributed architecture which in contrast to client-server has not a centralized service provider. Peers are both clients and servers themselves, P2P promotes sharing of resources and services trough direct exchange between peers. Compared to a centralized client-server architecture a P2P net scales better and typically does not have a single point of failure.

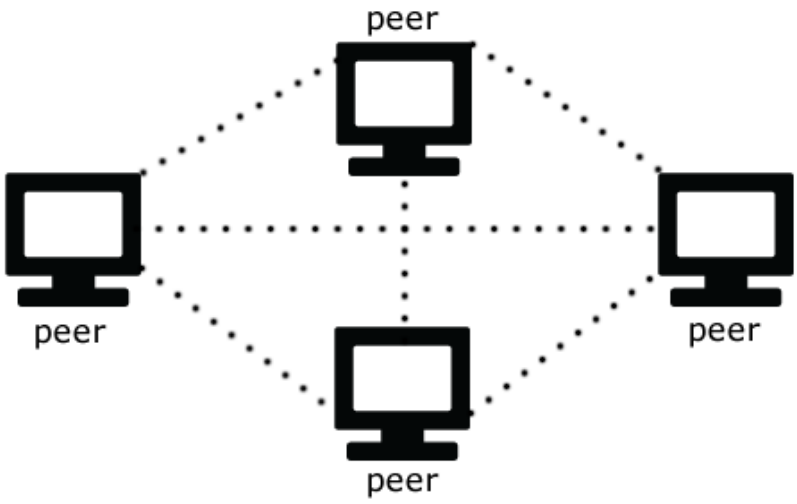


Figure 2.11: P2P architecture

REST style Representational State Transfer (REST) is a style of architecture based on a set of principles that describe how networked resources are defined and addressed.An application or architecture considered RESTful or REST-style is characterized by:

- state and functionality are divided into distributed resources,
- every resource is uniquely addressable using a uniform and minimal set of commands (typically using HTTP commands of GET, POST, PUT, or DELETE over the Internet),
- the protocol is client/server, stateless, layered, and supports caching.

Event based is an architecture in which components collaborate by exchanging information about occurring events. In particular components in the net can *pubblish* notifications about the events they observe or [subscribe] to events they are interested to be notified about. This architecture can be fully distributed with all the same nodes or can have some semi-centralized nodes which are specialized in computing events or routing messages. Communication is, in this case, purely message based asynchronous and multicast.

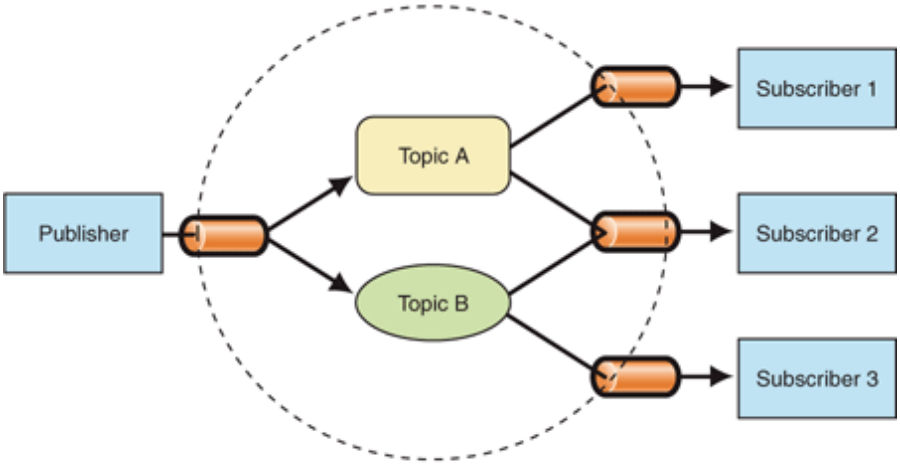


Figure 2.12: Publish-subscribe architecture

2.2.5 Examples

2.2.6 Liquid Computation

Chapter 3

Problem Analysis

In this chapter the specific problems of this work will be detailed and analyzed, explaining what are the limits and the constraints the challenge has. The chapter starts with a brief recap, followed by the proper definition of what I faced, while in the last part there is a list of constraints my architecture will have fulfilled in order to have a universal and functional solution.

3.1 Brief recap

In the previous chapter, number 2, I have defined IoT and WoT, pointing out the main differences between the two definitions. In particular I focused on the problem of the interconnection between the two different paradigms. The IoT can be seen as a network, in which not only standard computers are connected as nodes, but also smart objects: they are "first class citizens" of the internet, they have a connectivity module, they have the capacity to "talk" each other. But, due to some constraints, in particular power consumption constraints, it is not possible to consider the IoT as a big unified network, because devices have different communication protocols and not all of them are compatible with each other to offer communication. The result is a non standardized confused cloud of protocols, systems and technical stacks. Solutions are often proprietary and closed, even if some open solutions exist.
The WoT is the election of the smart objects to "first citizens" of the web, the web has a fundamental role as unifying layer, it is standardized, it easily reachable by everyone and there is no need of particular requirements on devices. The idea is bringing the devices to the web. The fundamental requirement is the implementation of TCP/IP and HTTP protocols. Not every object has the capacity to provide this connection, but most of them could, also with the help of a gateway which can be also called *reverse proxy*. In this chapter I am considering only devices that have this feature.
After this brief recap of what has been said about the Internet of Things and the Web of Things in the state of the art chapter, here I am trying to define with more precision the problem I am going to face: which its constraints and its possible goals are.

3.2 Definition

As anticipated above, I am going to take into account only devices that can be somehow connected to the internet, the ones which provides an implementation of HTTP protocol in particular, while in the substrates the stack can be left "opened": each vendor can implement its solution only if it provides a classical connection on the top. This statement can be seen as a "little relaxation" of the constraints previously announced.
Having done this clarification, now I am defining the problem.
As said the Web of Things is the "unification" layer built on top of Internet of Things, it does not replace IoT, it is a completion. There is a sort of vertical hierarchy. If both must coexist a sort of communication between them is fundamental to assure the right functioning of the entire stack. We have clear in mind what can be considered WoT for the final user: a sort of control center, reachable by a web browser, where anyone can control the smart objects he has rights on. Also what is IoT for final users is clear: it is a heterogenous world, a world made of objects they can control through smartphone, for instance. So we have an idea of the two parts we have to connect: on one side the well known web paradigm, with its programming languages, its limited number of web browsers; on the other a myriad of objects that have in common the possibility to act as web servers, managed through HTTP calls. *How can we let these two parts talk?* This is the question that my thesis is trying to answer. My work is a concrete solution, it is about creating or refining some communication methods.
So I am trying to let different worlds talk using well known tools: I have just said I am putting IoT and WoT in communication, but with the warranty that the smart objects can behave as web servers. So I am defining a method to connect different, "vertical" worlds, put one on the top of the other using a "horizontal" system: devices have HTTP endpoints that elect them to citizens of the web. But this does not guarantee they have a right behavior to be integrated in a heterogeneous and user friendly application. The result is a vertical connection that covers all the protocol stack, from the physical layer to the application one, while the last part of the connection is a "double step connection", a hardware part that guarantees the online presence of the device, and a software part that assure the right behavior and the standardization of the communication. Figure 3.1 on the facing page, describes what has just been explained: the green arrow indicates what is seen externally, the protocol stack, it links directly the application layer with the underlying ones. The red arrows indicate what I am proposing in this thesis: red one in the layer 4 is our requirement, the fact that the smart objects can act as web servers that are components of layer 7. The second red arrow, the one included in the application layer is the representation of what my work is trying to achieve, the software component of the full integration, as above described. As said the red arrow is all in the application layer, it remains "horizontal" in respect to the stack, it can be seen as an HTTP call done from the client, put in the "cloud" (note that what here is considered client can be the backend, the server, of a third-party application). The double side of the arrows stands, obviously, for the possibility to enable communication both sides, packing and unpacking information.

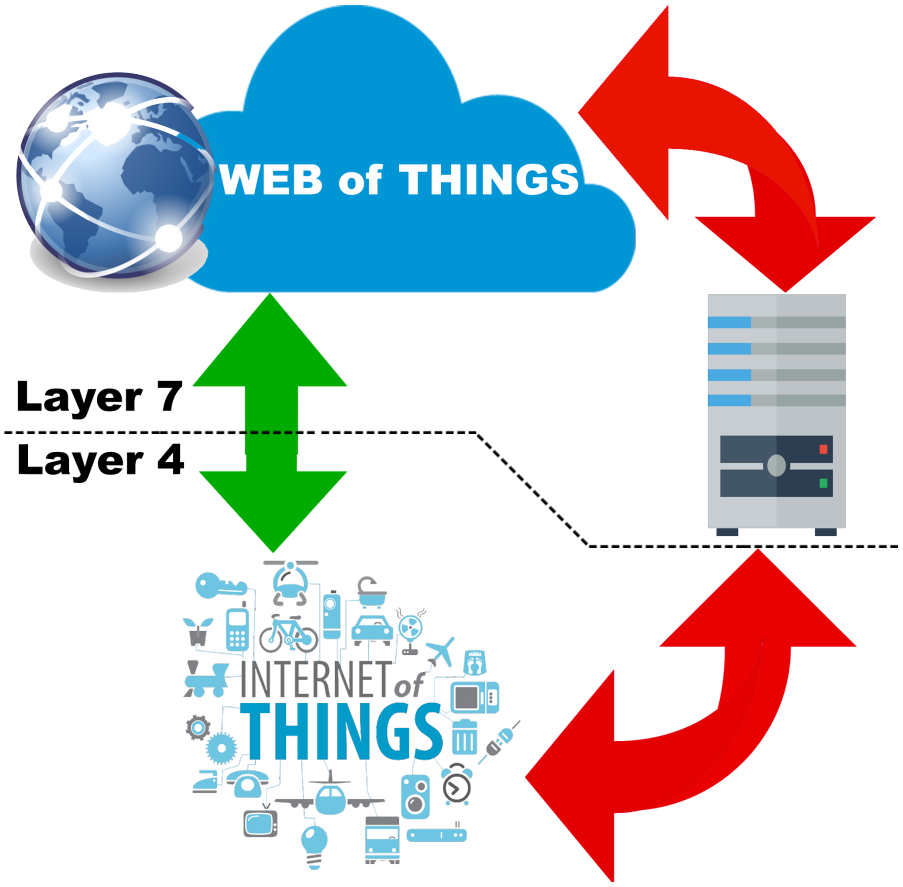


Figure 3.1: Stack result vs double step implementation

The important point is having a message with a well defined content: it is what the two parts must write and read, so it has to be clear for machines, must be compliant with all the requests of *M2M (Machine-to-Machine) communication*. On the other hand, a developer must be capable of reading and understanding what is going on in order to debug to solve errors or to write a customizable user application, so the communication must require also to match the *M2H (Machine-to-Human) communication* and *H2M (Human-to-Machine) communication* requirements. These types of communications are some constraints of my work and are explained in the next section 3.3.
So the problem I am going to face is the *lack of a standard communication method* between the smart objects and the web interfaces already present. What I am going to define in this thesis is a new "communication", efficient and practical. This can be seen as two distinct parts of the development: the choice of the "container" and the proper definition of the syntax. As said above, types of communications are a sort of constraints for the defined syntax, but also the choice of the "container" is not completely free, it has to match some fundamental constraints to assure the correct HTTP transmission of a message or a file.
Next sections will properly define all the constraints of the given problem and propose a solution that fulfills them all.

3.3 Constraints

In this section I would like to list a set of constraints for the defined problem, that become requirements that the solution must meet. The section should be divided into two parts, the first for the requirements of the container, the second for the ones of the syntax, but the two sections are closely related so here I preferred to keep the two parts united, indicating wherever possible which of the part that particular constraint acts on.
Here is the list:

- *M2M communication:* M2M communication is defined as a communication in which the two interlocutors are not humans. It is a communication completely handled by machines and computers [**cha2009trust**]. It can be considered one of the fundamental enabling technologies of the Internet of Things, it permits object to communicate without humans being involved. This type of communication has to be stricter respect to M2H: the subject reading the content is a computer, it has no a semantic idea of what is written, it can, at most, understand the syntax. So a clear, defined syntax with a well fixed structure must be set in order to make everything understandable to a computer. Obviously the machine has also to able to write the code or the file that will be sent. Anyway, the problem of what is understandable for a machine is a common "problem" to all fields in computer science and engineering.

- *M2H/H2M communication:* I grouped these two constraints because they are similar, they include some similar behaviors, in fact somehow the solution I am writing has to reach at least a human being, the owner of the smart space. In order to check what is going on and control the behavior of smart devices the user must understand what he sees on a web application. This constraint can be considered "indirect" for my work: it must be somehow adapted to a UI, the one of a control center application developed on top of the file I will provide. So it means that my work does not reach the user directly but it means that it should be possible to translate it easily into something the user can understand without too much effort. That means I have to use variables and values in a univocal way, in order to permit the web app a clean and meaningful translation. These considerations are valid for M2H communication. Regarding H2M communication the user must be able to easily impart commands and the inverse procedure of translation needs to be as simple as possible. So the user has some constraints: he cannot use free natural language, the problem is still the same, computers are not able to semantically understand natural language.

- *Easy readable:* What I am going to define is a sort of description of a smart space that must be readable by machines and by humans, as explained above. This does not imply the easy readable property: a thing can be read by human or machine but with difficulty. The desired property here is the fact that both humans and machines will be able to read the content of the smart space without much effort. This streamlines all the procedures. A

machine that has to execute a command or retrieve information has to do it very quickly, if something must be notified to the user, the content to (write and) read must not be an obstacle, in terms of speed or resources spent to assure the right communication. For humans, we can distinguish two categories, developers and users. For users the situation can be mitigated by the presence of a web application between my work and what he sees on the web pages. I will provide a description of objects which will be embedded into third-party applications. In this way the user does not see directly what I have done: developers are the people who use data provided by my system to construct a web application. Two are the main reasons they need understandable data, developers need data to be easy readable in order to easily integrate them in applications and also to debug the system. The error can be in the written code, but the source remains my work, having it clear in mind, it can help to quickly debug.

- *Reconfigurable:* we are talking about smart spaces, composed of smart objects, they are usually relatively small and so they can be moved easily (some of them, though may be bigger, such as industrial machinery). Another issue can be the fact that, being powered by batteries, objects can turn off due to low level battery status. These two just explained situations appear to my system as a change of of the smart space: it is necessary to reconfigure it, in order to output the current situation without having "dead devices" or moved ones into the described space. Moving a device, we can have the situation where a sensor, for example, is moved from a room to another, so the total number of devices is not changed but the configuration is. My system must be able to detect the "loss" of one of them in some place and the "gain" of the same one in another.

- *Fast:* being fast is one of the essential constraints of my work. I have just stated that the reconfigurable property is desired, to handle the change of the spaces my thesis is going to describe. But how fast my system reconfigures all the list of objects becomes fundamental. No latency is permitted. Think about an alarm or a thermometer which goes out due to the level of the battery: the user must be notified and aware as soon as possible. The user cannot be kept without any protection for his home or without heating in winter. There should be a mechanism, a sort of listener, that understands when something is changed and reports quickly to the third-party application the user uses.

- *Lightweight:* Another constraint to our system is the fact that whatever system I choose to be the solution it must be lightweight. This is needed because it goes on the web, the third-party application server requires information from the objects that it collects, packs it and sends to the user. The user can see its configuration through a web browser that, as a client, downloads the required page. If we want information to be notified quickly, information should necessarily be lightweight. So my solution has to take into account the fact we are talking of the Web of Things, and the word "web" implies some constraints by definition: the one we are interested in

here is the latency and the transmission time, due to the physical limits of the cable or the materials that constitute the network infrastructure.

- *Modular:* the language or type of file chosen must be modular. The faced problem is modular itself: it is necessary to group lots of IoT devices into one singular description. It must be a description of the smart space: it is composed, for instance, of rooms. And also in each room there can be a hierarchy of devices, in addition the objects with a gateway need to be described as coupled, and only one gateway can have multiple devices to control. So the solution also has to be modular, it must be possible to write different configurations. It is necessary to define some building blocks, that can be placed almost everywhere in the hierarchy. The length or the number of levels that can be nested should be variable, each smart space has different particularities: starting from a city to a living room. My solution must be universal: a thermometer, for instance, can be put in both of the listed places without specifying where it is and without changing internally its properties due to the position.

- *Complete:* the chosen solution must be complete, in the sense that it must be able to write down every configuration. It should be possible to describe any kind of object, it has to own each kind of smart object present on the market, with their properties, status and commands. It should also be possible to insert each object in the hierarchy almost everywhere, even if, obviously, some rules are needed to assure the right syntax and the right structure. A "language" that is not complete cannot be accepted, it must be able to describe any given smart place, my solution tries to be a unifying method for developers that can count on it as a consistent description of the space, the "source" of what they have to output to the user.

- *Extensible:* the solution must be extensible, because every day new objects become smart. With the technical innovation it will be possible to increasingly miniaturize the chips needed to offer a connection, to become smart. My language needs to be extensible, when a new sort of product comes on the market my language has to define a standard for it, according to the object's properties, states and features. It must be always possible to do that, there should not be limits to the number of building blocks my solution is going to define, because we do not know today where the "smartness" of objects is going to go, we do not know the level of miniaturization and power saving that will be reached in the future. Being extensible means being a valid alternative also in the future, writing a solution that only fits "today" objects without the possibility to expand limits my work to be just a "photograph" of the current state. With the growth of the number and types of smart devices, the extensible property becomes fundamental to obtain another property: completeness.

The listed requirements, as already told in some of them, are, sometimes, general, in the sense that they have to be respected for the final product: a global and complete structure that starts from the smart object and arrives to the user's

web browser. This is because the problem I am facing is very big and complex, and it is transversal to the existing technologies, so the whole system must work properly. Keeping in mind what I have just stated, some of these constraints become fundamental requirements that my system must meet, some others, for instance the M2H communication, are only indirect or partial requirements: my work has to be clear for the developer, who knows what variables are, how a file can be accessed, parsed and processed but it can be less clear to an average user who wants, out of curiosity, see what is under the hood.

In the next chapter I am presenting my idea, the so called solution to the given problem, explaining what I have done, my considerations about the situation here faced. I am defining both the container and the syntax, or better the structure, of my work; these two parts are the fundamental ideas that are the bases to construct my thesis.

Chapter 4

Proposed Solution

In this chapter the development of the solution will be reported step by step, with some use cases. The chapter starts with a list of solutions already developed, already on the market and he differences between them and my work. The remaining part is composed of two mains sections: the first explains the choice of the so called *container*, the architecture, the type of file, etc. It lays the foundations and describes the limitations for the second part: the definition of the *syntax*, or better the *structure* of what is written inside the container. The two parts are closely related, therefore their relation was taken into account when I made my choice.
The real implementation of the valid solution is left for the next chapter.

4.1 Already developed solutions

A lot of companies have already tried to create a solution as "universal" as possible, but as their main aim is profit they are not really interested in creating a "real" universal solution. Their main goal is including more and more smart objects and making them compatible with their systems. The Web of Things for them is not an arrival point to reach, but rather an obstacle for increasing their profits. Companies are not innovating in the sense of creating a common framework that every vendor can use. They are trying only to enlarge their sphere of influence, not to be a real change in the world. This means companies aim to keep the IoT as final product to offer to users, doing it with some partnerships with the companies which produce smart devices. It is only a matter of time before some exclusive contracts will be signed: if some producer becomes a supplier for only one system, there is no gain for the community, a user has to choose which objects to buy depending on the system he has maybe already chosen and not depending on the specifications or the features of the device.
For instance we can analyze some products, such as *Apple HomeKit*. Apple has built this groundbreaking application with the possibility to integrate devices to control a smart space, in particular a house. The idea is good, the application, in my opinion, is fantastic, but there are some limitations: the application can run only on iOS devices, and this constraint is somehow strict, as already explained in the first two point of the **??** section. Then, as explained here, the only devices

that are compatible are the one which includes a particular protocol, always developed by Apple [**applehomekitprotocol**]. It is called *HomeKit Accessory Protocol (HAP)*, it is closed and vendors have to implement it inside the objects. This second requirement is stricter than what I have stated in 3.2 section, I only "impose" objects to have an HTTP connection and act as web servers: there is no simpler request to be on the web. Here it is different, HAP can work also on Bluetooth Low Energy (LE), but an upper layer is needed.

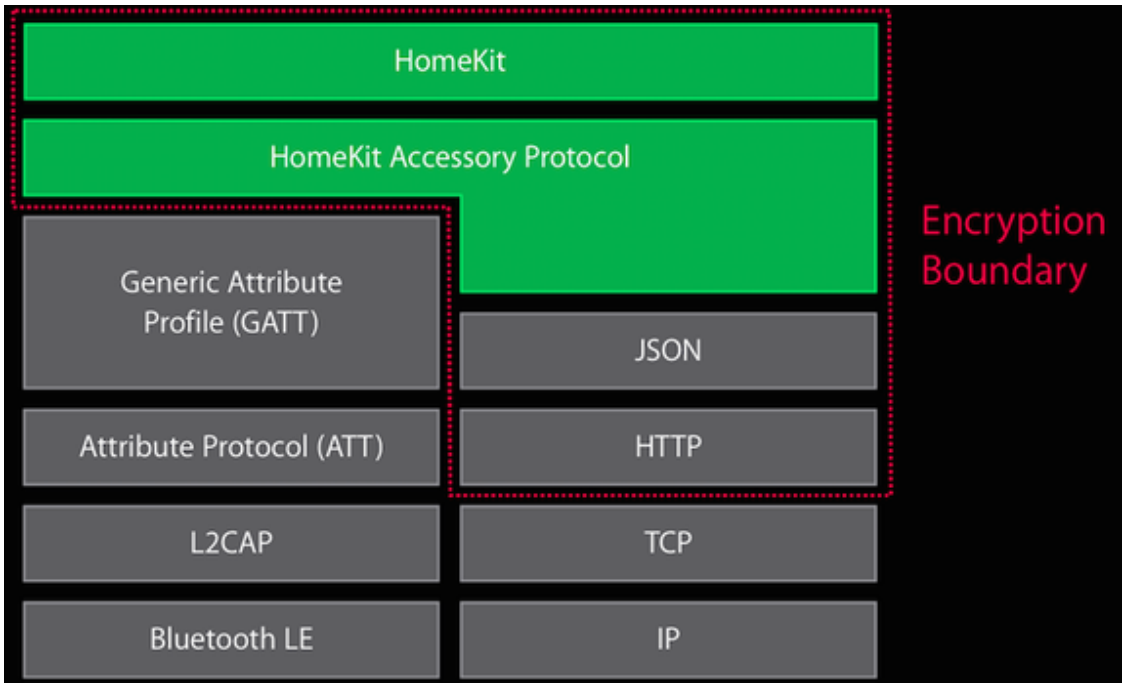


Figure 4.1: HomeKit Accessory Protocol (HAP)

I quoted this solution only to give the reader an idea of what is already existing, but many other systems are present on the market, Google has its system, called *Google Brillo* that works similarly. Brillo is an Operative System (O.S.), obviously based on Android, which must be installed on the smart object. It "violates" my requirements about the liberty of a device concerning hot to reach the web and how to implement underlying layers. Also Microsoft has its solution, it enjoyed the *Allseen Alliance*, an open solution that embraces all the levels of the protocol stack, using ad hoc developed layers.
All the solutions I have very briefly listed here are only some of the ones the market today offers. But they have a different purpose, they wants the devices mounting an operative system, a protocol, etc. This still remains in the situation explained in chapter 2, a war fought by different companies or alliances, leading the world to a myriad of different and incompatible solutions: it is more an attempt to enlarge each own IoT, respect than the creation of an open solution for a common widely resource like the web. Now it is the moment to start with my reflections about the given problem comparing the available paths to reach a solution.

4.2 Choice of the *container*

To better explain what I consider container it is important to understand the playground to my work. As said in the previous chapter, 3, I am trying to fulfill the "horizontal" part of the connection in the stack, staying in layer 7, the application layer. Using already developed and operating tools, and respecting all the above listed constraints I am going to make the communication between WoT and IoT "HTTP ready" devices possible. I will now present alternatives to assure communications, choosing the one that better fits my requirements. The choice of the so called container is very important: it puts boundaries in the structure/syntax definition, so it is essential to understand pros and cons of each candidate.
I am searching for technologies that permit clients and servers to communicate, operating on the same network. The network is composed of three main actors: the user's client which is the web application opened, for instance, on a smartphone or a pc; the web server which it is the real application and it is on a machine which it behaves as a server for users and as a client for the smart objects; the objects, that have an end point on the HTTP protocol and behave as servers for the web server which asks information periodically.
What we need is a *Web Service (WS)*. To understand what a WS is, there is the necessity to understand what we are doing and what we want to accomplish.

4.2.1 Web Service (WS)

The web today is a sort of container of more than hyperlinks, it is a distributed application platform. If it is distributed data should be exchanged among nodes. What enables this exchange is the Web Service. A WS can be defined as a software component that permits data to flow among different applications (written in different programming languages) that runs on different operative systems, installed in different nodes of the network [**alonso2004web**]. This mechanism does not put any limitation on the type of the application developed on the web and it is perfect. It does not affect the internal implementation of a web application, it is completely independent. A web service exposes some services, that can be "called" remotely to the other nodes of the network These services are often called *APIs*, that stands for *Application Programming Interfaces*, and they are particular functions that the developer of an application chooses to make reachable from others. It is possible to call the APIs through messages, more specifically W3C has standardized only one way to do it, through HTTP protocol. The WS has obviously some advantages and disadvantages that I will list.
Advantages:

- Interoperability among different software applications that run on different hardware platforms is possible.

- Data format can be considered "textual", that is easy readable both for machines and humans.

- Using HTTP for the transport of the message Web Services do not need any change in security rules of the firewalls of the nodes of the network.
- Web services are completely independent one from another, so they can be combined in a more complex and complete service offered to the user.
- There is no need to rewrite applications already existing to make them compatible to WS. They are completely independent also from modification done successively.

Disadvantages:

- Consolidated standards for critic applications, such as distributed transactions, do not yet exist.
- Some other alternative approaches for distributed computing can have better performances in some situations (Java RMI, CORBA, DCOM). They are briefly discussed later.
- They can "avoid" firewall controls and that can potentially be dangerous.

The main reasons of the wide adoption of Web Services are, as said, firstly, the "decoupling" between the service itself and the application behind it, without worries about modifying or rewrite one of the two sides, they are completely independent. Secondly, it uses HTTP over TCP on the 80 port, the one left (almost) always opened in each system so as to enable to surf the net even company networks that often have limitations for security reasons.
The Web Services have two main different implementations, *SOAP* and *REST*. These two approaches will be examined in the next subsections, now I will briefly focus on the alternative methods respect to these two.
The alternatives to Web Services are not taken seriously into account because they cannot be considered universal, as they are too closely related to a specific language or a set of languages. For example, *Java RMI* stands for Remote Method Invocation and it enables to invoke a function on a remote object. Methods must be public and known by everyone. But it works only among Java sandboxes, so the application must be written in the Java language. It is a strict requirement for the whole web and it cannot be used in my work. *CORBA (Common Object Request Broker Architecture)*, for instance, is another standard that can work with a set of different languages, which means not all the languages are accepted, but an intermediary is needed, called broker, which reads objects written in a special language that is IDL (Interface Description Language). And the translations between IDL and the specific languages are what makes CORBA compatible only with a defined set of programming languages. In any case this particular technique requires all the nodes of the networks to support it, an HTTP connection on 80 port is more and more simple.
These two examples, obviously, do not cover all the alternatives, but they are quoted to explain how the Web Services and their features are the most frequently used and effective solution. The next sections examine SOAP and REST and explain how and why I took my decision and reached my solution.

4.2.2 SOAP protocol

SOAP (Simple Object Access Protocol), as the name says, is a protocol that permits to call remote procedures (procedures are methods). This is enabled by an exchange of SOAP messages. SOAP is a member of the *Remote Procedural Call* protocols family. SOAP is a standalone protocol that must be embedded in other protocols in order to surf the network and reach the proper node. The *W3C* has standardized only HTTP to be the protocol which embeds SOAP, even if it is possible to embed also in the STMP (Simple Mail Transfer Protocol) one. SOAP can be viewed as a protocol that stays on top of the HTTP one and uses it as transport to reach every part of the network. It is older than its "opponent" REST, it was originally created by Microsoft and then submitted to the Internet Engineering Task Force (IETF) where it was standardized [**soap**].
Once it is understood that the fact that SOAP uses (mainly, but it is my case so I consider it a "standard") HTTP as a transport layer, we have to analyze the content of the message SOAP sends and receives. SOAP messages are based on the XML language that will now briefly explained.
Starting from scratch, *XML (eXtensible Markup Language)* is a language of markup, it means it has a syntactic mechanism to define and control the meanings of what is written inside. It defines the format of file .xml that can contain information which is possible to exchange on the network. It is composed of a hierarchical structure of tags, which can contain information (called also #PCDATA, "Parsed Character Data") or other tags, it also is considered extensible because it is possible to create custom tags of any type only declaring them. As a small example, I report the following XML file, describing a list of users which can be, for example, the export of the contacts on a smartphone.

Listing 4.1: XML file example

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
    <user>
        <name>Jane</name>
        <surname>Doe</surname>
        <number>+3933333333</number>
    </user>
    <user>
        <name>Jack</name>
        <surname>Sparrow</surname>
        <number>+3944444444</number>
    </user>
</users>
```

The first line is called prologue and it is only a declaration of the XML version used and the encoding UTF-8 to interpret data correctly. The root tag, which is unique, is "users", that is a list of "user", each user is composed of a name, a surname and a number. The tags are formed by <> with in between the name of the tag. In this case tags are balanced: a tag is opened, then there is the content

and finally it is closed. Also the nesting of tags is respected: opening "user" and then "name" I have to close the last one opened so I close "name" and then "user". These three properties (prologue, root that must be unique, balanced tags), if present, define a Well Formed XML. Having a WF XML it is possible to add a *DTD (Document Type Definition)* file that contains rules to write a particular type of XML. DTD is out of our way, so I will not explain it in details. The only important thing is that it contains rules for writing XML, defining which elements can have sons, and type of information contained (link, string, etc). Another more recent type for the definition of the structure of an XML file is the *XML Schema*, called also *XSD, XML Schema Definition*. It is properly an XML file that defines a XML file and it supports datatypes (while DTD does not), and more generally it supports more options for defining precisely all the fields of the XML. If an XML file is Well Formed and it follows a DTD or XSD specifications, the XML file is called Valid.
In particular, a SOAP message is composed of a root tag called "soap:Envelope" that has two sons: "soap:Header" and "soap:Body", similar to the HTML structure. The header is optional while the body is mandatory. The header contains meta-information regarding routing, security, user identity, transactions etc. Two particular fields are remarkable: "MustUnderstand" that must be put equal to 1 if we want to oblige the reader to parse and read the header, because it contains vital information, for instance how to decipher a message. "Actor" contains the address of the recipient, so the intermediate endpoints can avoid to read the header. The body contains information, called payload. The payload has to follow a schema defined with a XSD, in order to assure that the SOAP message is Valid. The structure of the SOAP message just described is completely independent from the underlying protocol, HTTP, STMP, etc. that is used for transport [**soapMessage**].

Listing 4.2: SOAP message example

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <soap:Envelope
3          xmlns:soapenv="http://schemas.xmlsoap.org/soap/
              envelope/"
4          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5          xmlns:xsi="http://www.w3.org/2001/XMLSchema-
              instance">
6
7    <soap:Header>
8      <ns1:RequestHeader
9            soapenv:actor="http://schemas.xmlsoap.org/soap/
                actor/next"
10           soapenv:mustUnderstand="0"
11           xmlns:ns1="https://www.google.com/apis/ads/
                publisher/v201605">
12       <ns1:networkCode>123456</ns1:networkCode>
13       <ns1:applicationName>Babel Framework</
             ns1:applicationName>
```

```
    </ns1:RequestHeader>
  </soap:Header>

  <soap:Body xmlns:m="http://www.xyz.org/quotations" >
      <m:GetQuotation>
         <m:QuotationsName>Apple</m:QuotationsName>
      </m:GetQuotation>
   </soap:Body>

</soap:Envelope>
```

The above file contains an example of a SOAP message that requires the value of Apple quotations to another node in the network. It is possible to identify all the elements previously described.
But, how can it be possible to know the right syntax of a tag to put in the "soap:Body" to have back a precise element? The *WSDL (Web Services Description Language)* describes how the Web Services work, WSDL can be seen as a public interface of a Web Service [**curbera2002unraveling**]. The WDSL is an XML file separated by SOAP messages, that is a sort of "handbook" of how to use the Web Service. It is mainly composed of three sections: "what" can be used, which means the operations that is possible to call through the network; "how" it is possible to use the service: the type of the protocol to use, the type of both input and output messages and the bindings of the service; "where", the endpoint to which it is possible to require the service, it is an URI. The WSDL can be divided also in another way: logic and concrete sections. The logic part contains interfaces, operations and messages, while the second defines transport, bindings and endpoints. The current version is the 2.0 released in 2007 and it is a standard for the W3C. In the next listing I will give a example of WSDL, it is only an simple example to provide the reader with a more concrete idea of what we are talking about.

Listing 4.3: WSDL file example

```
<definitions name="HelloWorldService"
   targetNamespace="http://www.examples.com/wsdl/
      HelloWorldService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.examples.com/wsdl/
      HelloWorldService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="SayHelloWorldRequest">
      <part name="firstName" type="xsd:string"/>
   </message>

   <message name="SayHelloWorldResponse">
```

```
13        <part name="greeting" type="xsd:string"/>
14    </message>
15
16    <portType name="HelloWorld_PortType">
17        <operation name="sayHelloWorld">
18            <input message="tns:SayHelloWorldRequest"/>
19            <output message="tns:SayHelloWorldResponse"/>
20        </operation>
21    </portType>
22
23    <binding name="HelloWorld_Binding" type="
          tns:HelloWorld_PortType">
24        <soap:binding style="rpc"
25            transport="http://schemas.xmlsoap.org/soap/http
                "/>
26        <operation name="sayHelloWorld">
27            <soap:operation soapAction="sayHelloWorld"/>
28            <input>
29                <soap:body
30                    encodingStyle="http://schemas.xmlsoap.org
                        /soap/encoding/"
31                    namespace="urn:examples:helloworldservice
                        "
32                    use="encoded"/>
33            </input>
34
35            <output>
36                <soap:body
37                    encodingStyle="http://schemas.xmlsoap.org
                        /soap/encoding/"
38                    namespace="urn:examples:helloworldservice
                        "
39                    use="encoded"/>
40            </output>
41        </operation>
42    </binding>
43
44    <service name="HelloWorld_Service">
45        <documentation>WSDL File for HelloService</
              documentation>
46        <port binding="tns:HelloWorld_Binding" name="
              HelloWorld_Port">
47            <soap:address
48                location="http://www.examples.com/
                    SayHelloWorld/" />
49        </port>
50    </service>
```

```
51 </definitions>
```

Giving a brief description of the WSDL file, the tag "message" contains the type of message that the service has both in input or output and "name" indicates the parameter of the message, the tag "portType" describes the full available operation, indicating also which is the input message of the service and which one is the output, these two tags together are we what have called "what" previously; the tag "binding" is the "how", it explains that the service is reachable with the SOAP protocol and specifies the body of the message; finally, the tag "service" indicates the endpoint to which is possible to request the service, it is the "where" section previously defined.
Back to SOAP, one thing remains to be said, it is highly extensible, in particular some modules can be added: the term *WS-** stands for Web Services, but the * indicates all the possible ones. An extension is a variation of "something" in the SOAP message to emphasize a specific feature. Lots of extensions exist: for instance, WS-Security (WSS) is one concerning security, it can be used for some particular application where an identification or even authentication is required, it becomes less useful for open application. It guarantees, for example, end-to-end security, identification tokens and many other features.

4.2.3 REST paradigm

The SOAP protocol and all its "environment" are only the first alternative we have to enable Web Services. The second one is the architecture called *REST (REpresentational State Transfer)*, it was conceived for systems of distributed hypertext, as the web is today [**fielding2000representational**]. In fact the World Wide Web is the most successful field in which the REST paradigm is used, but not the only one. The first thing to specify is that while SOAP is a particular, well defined protocol, standardized with a specific set of rules to follow, REST instead is a set of guidelines, a paradigm as just said, for the realization of a "system architecture". The word REST, indicating a "way to do things", can be considered the opponent of the term Remote Procedure Call (RPC), that indicates the procedure previously described, rather than SOAP, but, a unique implementation of a RESTful device does not exist I am putting them on the same level here. Note that a service implementing the REST paradigm is called *RESTful service*.
Basing our analysis only on the web it is possible to state that REST is based on the concept of transmitting information over HTTP without the use of an upper optional layer such as CORBA, SOAP or the cookies used in websites nowadays. REST paradigm is based on principles that the web today knows very well, but they are somehow reinvented to be part of a Web Service. Firstly we have to define what can be considered *resource* for the web:

- A resource is every web element that had an elaboration. It is possible to compare a resource to an instance of an object in the Object Oriented programming paradigm.

- A resource can contain the state of an application or the functions it provides.
- A resource is uniquely identifiable by a string called *URI (Universal Resource Identifier)*, a URI can be seen as the link to put together more than one resource.

Having defined what a web resource is, the REST paradigm follows some other principles to exchange information among nodes of the network. The resource is what permits the sharing, but even the exchange process has to follow some guidelines respect to REST principles. The resources are shared as a common interface to allow the transfer of states among resources. The main features of the transfer are:

- a fixed set of well known operations.
- a fixed set of contents, maybe with a particular request code.
- a protocol, that itself has to be:
  - client-server: the division is strong, every part has its role, for instance, the client has no worries about saving information while the server has no worries about the graphical interface. If the interface that links them remains the same it is possible to modify them separately.
  - uniform interface: client and server need a homogeneous interface. The two parts can evolve separately from the interface.
  - stateless: no context is saved in the server, every request coming from the client has all the information needed to serve it.
  - cacheable: the client can cache answers, but they have to declare if they are cacheable or not in order to prevent the use of wrong and invalid old answers. On the other hand the cache can reduce the number of client-server communications, increasing performances.
  - layered system: there is no only one server, it is possible to introduce some middleware servers to improve scalability with load-balancing or distributed caches. In addition the multi-layer models enables to introduce more secure mechanisms, isolating the information from external attacks.
  - code on demand (optional): servers can send executable code that the client will interpret or compile, such as JavaScript.

Now that the main features have been presented, I will briefly informally describe the full procedure of a RESTful service. Knowing a source of information (the resource), through the URI, the client can make a request and the server can "listen" what is requested. This communication happens through a standard interface (for example HTTP protocol). In this way representations of the resource can be exchanged: receiving the request the server can answer in multiple ways, without knowing the "past story" between the two parts. There can also be

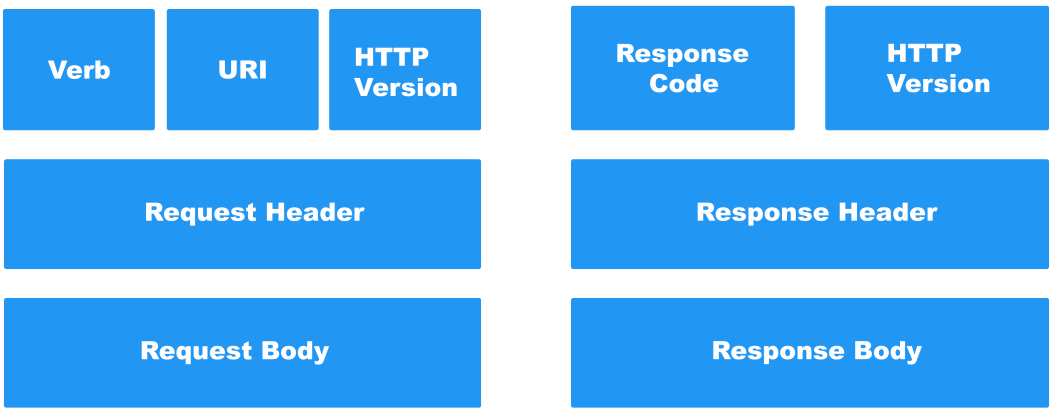
the intermediation of middleware components, proxies, firewalls, etc, this does not affect the final result. The only thing the client needs to know is "what" is returned by the server: the format of the file is very important, the client needs to be able to process it once received. Typically the format is a HTML, XML or JSON file containing the information but answers, for instance, may also be given with a picture.
The use of HTTP makes requesting and interacting with the information required very simple: this protocol provides a set of primitives, functions implemented directly inside that can be used by everyone. REST uses four of these verbs: *GET, POST, PUT* and *DELETE*. For example, the state of service can be obtained performing a HTTP GET to the URI, while in a not RESTful service we can custom methods, with arbitrary names depending on the implementation. In addition, one of the most important things is that HTTP verbs are 1:1 mapping to *CRUD* operations. CRUDs are the fundamental operations that a user can execute on a resource. (More generally four actions must be implemented in a RESTful application, in a relational database and in a document management application to consider them "complete").

Table 4.1: Mapping between CRUD operations and HTTP 1.1 verbs.

HTTP Verb	CRUD Operation	Description
POST	Create	Create a new resource
GET	Read	Obtain an existing resource
PUT	Update	Update or change the status of a resource
DELETE	Delete	Delete a resource

In this way the HTTP protocol can cover all the possible actions a user can perform on a resource. This is one of the strengths of the REST paradigm.
At this point we have described what REST means, how it works and how it covers the operations needed to handle a resource. Now we will focus on the representation of this last component: what comes to the client is never the resource itself, it is always a representation. As said before there are no constraints on the type of file that is going to represent the resource on the client side, anyway it is better to uniform or, better, circumscribe the number and types of file that a RESTful service outputs. If the servers offers more than one representation it is possible for the client to choose one, indicating it in the "Accept" field of the request message. For example, a browser, when we go to a particular website, is doing an HTTP GET with the predefined "Accept" field equal to "HTML". In this case we have a Web UI, with all the graphical elements, pointing to the same page, changing the "Accept" to "JSON" we have a Web API. Following the REST principle, we can construct different architectures using the same URI, the same resource but changing its representation.
In the following pictures I am going to quickly describe HTTP messages, requests and responses, to have a full comparison with SOAP messages. Then a description of each sector of the message is provided.

The HTTP Request message is composed of main 5 sectors:



(a) HTTP Request **(b)** HTTP Response

Figure 4.2: HTTP Messages

- Verb- Indicate HTTP methods such as GET, POST, DELETE, PUT etc.
- URI- Uniform Resource Identifier (URI) to identify the resource on server
- HTTP Version- Indicate HTTP version, for example HTTP v1.1 .
- Request Header- Contains metadata for the HTTP Request message as key-value pairs. For example, client (or browser) type, format supported by client, format of message body, cache settings etc.
- Request Body- Message content or Resource representation

An HTTP Response message has four major parts:

- Status/Response Code- Indicate Server status for the requested resource. For example 404 means resource not found and 200 means response is ok.
- HTTP Version- Indicate HTTP version, for example HTTP v1.1 .
- Response Header- Contains metadata for the HTTP Response message as key-value pairs. For example, content length, content type, response date, server type etc.
- Response Body- Response message content or Resource representation.

The stateless communication is done to maintain scalability inside the server, an open connection or session is a waste of energy and memory space: if too many requests come together the server can crash, it is the base of a DDOS attack for not RESTful services.

4.2.4 Comparison and choice

We have discussed in an appropriate way the two main candidates for the given problem. Here I would like to list pros and cons of both solutions and then choose which container to use, in particular the pros of a system can be considered the lacking items of the other, in the sense that if one feature is on a list it means that it is exclusive of that architecture:

- REST is generally easier to use and is more flexible:
  - No expensive tools require to interact with the Web service
  - Smaller learning curve
  - Efficient (SOAP uses XML for all messages, REST can use smaller message formats)
  - Fast (no extensive processing required)
  - Closer to other Web technologies in design philosophy
- SOAP is definitely the heavyweight choice for Web service access:
  - Language, platform, and transport independent (REST requires use of HTTP)
  - Works well in distributed enterprise environments (REST assumes direct point-to-point communication)
  - Standardized
  - Provides significant pre-build extensibility in the form of the WS* standards
  - Built-in error handling
  - Automation when used with certain language products

Taking a look at the points listed above, I would remark how the pros of the REST paradigm match some of the requirements already listed in 3.3. Efficient and fast were two of the main characteristics needed by our system. Then the fact no expensive tools are needed is another important point, developing a sort of framework for third-party applications. Another advantage of REST is being "language independent". In addition it can be chosen only as "container", leaving the choice of the type of file free, even if we stated that it is better to limit the formats of file sent on the web. As far as SOAP advantages are concerned, it is important to underline how the fact it is independent from HTTP (while REST is dependent) is not a problem because we are talking about the Web of Things, we are supposing to put everything on the web, and the HTTP protocol is one of web's fundamental parts. The only point that can tilt in SOAP direction is the fact that SOAP is standardized while REST is not, it is a set of guidelines, not a well defined structure [**pautasso2008restful**].
As the reader can guess, the choice for my system is the REST paradigm. Despite the fact it is not standardized, REST provides a set of easier "rules" to build and develop a framework easily. The not-standardized part is what I am going

to define in the next section of my thesis. Basing my own architecture on the HTTP messages and REST paradigm, the choice of the file surfing the web is still undecided, and the universal syntax I propose is still to be created.

4.3 Structure definition

This section is intended to define the format of file of my REST solution, and then the syntax inside it. As previously said, every format of file can be embedded into HTTP body and sent on the internet but it is a good habit to limit the number of circulating formats. In particular we can identify what is needed for our problem, restricting the choice.
We need to send to an application, a control center, the description of a smart space, in order to permit the user to control everything. The heterogenous elements are collected with a sort of "discovery", while in my application some "plugs" for particular chosen protocols are created.

4.3.1 Available alternatives

So we have to send descriptions of objects, the descriptions are composed of strings and numbers. It is possible to identify three main formats to send this type of information: HTML, XML and JSON.

- *HTML:* the first format is the classical building block of a web page, it is a language that contains graphical elements, it is created for the web and it is in line with our intent but it is intended for humans, in particular for M2H communications, the final recipient is a human being. This doe not match our requirements: we need, as explained in 3.3, a easy readable language but principally for a machine, my work needs to be a framework, something that links objects of the Internet of Things and the final UI the user will see. For this reason I am going to eliminate from the list of eligible candidates the HTML opportunity. It is the developer that will use my work as input to use, almost surely, HTML as language to speak to people.

- *XML:* this language is approximately suitable for my purpose, it has all the credentials to be the choice, but it can be considered so much verbose and heavyweight compared to JSON. Anyway I am not going to deepen XML because it is already been done in 4.2.2, explaining what kind of language SOAP uses.

- *JSON:* JSON stays for *JavaScript Object Notation*. It is a language created for data exchange in a client-server architecture. It is the choice I have made, and in the following subsection there is a full description what it is, how it works and above all how it can fit perfectly my requirements for the thesis.

I chose JSON for its lightweight, simplicity in writing and reading and also because there is a proposal of W3C to elect JSON to the official language for IoT

and WoT. Firstly, I will discuss all the features of JSON, then I will do the same thing for one of its extensions *JSON-LD*, I will present the W3C proposal and finally my solution, remarking what changes respect to others already developed.

4.3.2 JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, standardized in December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages. These properties make JSON an ideal data-interchange language. [**jsondef**]
It is based mainly on two types of structures:

- *Key/Value pair set:* it can be considered an object of an Object Oriented programming language. It is "contained" by a left () and right () curly brace, each element of the set is separated from others with a coma (,) and it is composed by a *key*, that is a string, followed by semicolon (:), followed by the *value*.

- *Collection of elements:* it can be considered an array, but it covers only *values*, in the sense that in an object, we can define a key, write the semicolon but the values associated to the key are multiple, so they are put in an array. Syntactically they are composed of a left square bracket ([), values separated by commas (,) and after the last value there is a right square bracket (]).

It is important to know what can be assigned to the *value* field. A *value* can be an array, as just said, a string, a number, a *boolean value* (true or false), *null* or another object, constructed as above. This means that nested structures are allowed in JSON format. String are wrapped in double quotes (""), using backslash escapes, empty strings are allowed. The concept of the string here is the same of Java, for instance.
While XML can be considered a markup language, JSON is more a format created for exchange of data. In both language there is no concept of binary data, so the developer has to convert data from the binary format while writing (eg. an integer into a "textual form").
As for XML, a "schema" exists: for XML is called XSD, while here simply *JSON data schema*. It can be used to validate JSON data. As in XSD, the same serialization/deserialization tools can be used both for the schema and data. The schema is self-describing. Anyway it is not a standard like XSD, but only an "Internet Draft (I-D)" proposed by IETF (Internet Engineering eTask Force), in order to become standardized.
In the above picture I am going to report in graphical way what has just been described, the syntactic rules to compose a JSON document, in order to give the reader a easier way to understand the format.

```
object                    elements
    {}                        value
    { members }               value , elements

members                   value
    pair                      string
    pair , members            number
                              object
pair                          array
    string : value            true
                              false
array                         null
    []
    [ elements ]
```

Figure 4.3: JSON syntax rules

JSON format is both *M2M* and *M2H/H2M* constraints compliant, and also *easy readable* (and writable). Having these properties we can also state is *fast*: both machines and humans are quick in doing actions on it. It is also *lightweight*, in particular respect to XML, being less verbose and less depending on open-closed tags. Finally, as it is possible to see from the above picture it is *modular* and *extensible* with the possibility to nest levels.

4.3.3 JSON-LD

JSON-LD stays for *JavaScript Object Notation for Linked Data* and it is intended to be an extension, an evolution to the JSON: it brings the web to a semantic level. To understand what is the *Semantic Web*, I have to firstly explain what are *Linked Data*.

Linked Data and Semantic Web The internet is seen as a network composed of nodes that are "bare metal" objects: computers and servers principally, or smart objects. The web, in the same way, is a network, but it is composed of nodes that are websites or web applications. These nodes are linked, with URIs, as widely explained. While the humans can understand what a link stands for, based on where is put in a page or which keyword is put near it. The machines cannot have this idea, for them a link is simply a reference to another "portion of the web", another web page to download. To make the machines understand what a link is and what represent in that particular context, there is the need to construct linked data. Linked data are simply a way to make the machines conscious of what they are facing: in this way also computers can have a global

vision of the web as a graph, almost connected, understanding the relations existing among different pages or applications. If, for example, we define on the web the concept of human being and the relation "marriedTo", the computer can know that the the object of the relation is another human being, so it can expect some type of data and it can raise an error if, for instance, the received type is a cat. This understanding can lead, for example, a computer to be more and more precise and capable of answering complex queries like "Who is Thomas father?": knowing the relation "father", the machine has only to follow the link provided to get and show the answer. A semantic cognition is what modern search engines are trying to reach.
Meta-data is what permits the recognition, like a index for a library: there you can find all the useful information to catalogue the real information, the books. Here is the same, meta data, specified and filled in the HTML page are what permits the computer to know the relations existing among what it has already downloaded and what it can reach from there.
If the web, a complicated graph, has the possibility to be understood by a machine, we are talking about semantic web. A web with a with a consciousness of itself. [**bizer2009linked**]
I will now report a figure, explaining graphically the concept of semantic web, using *RFDa (Resource Description Framework in Attributes)* syntax. RFDa is a W3C Recommendation that adds a set of attribute-level extensions to HTML, XHTML and various XML-based document types for embedding rich metadata within Web documents. Anyway RFDa is beyond our purpose so it is not explained in depth.

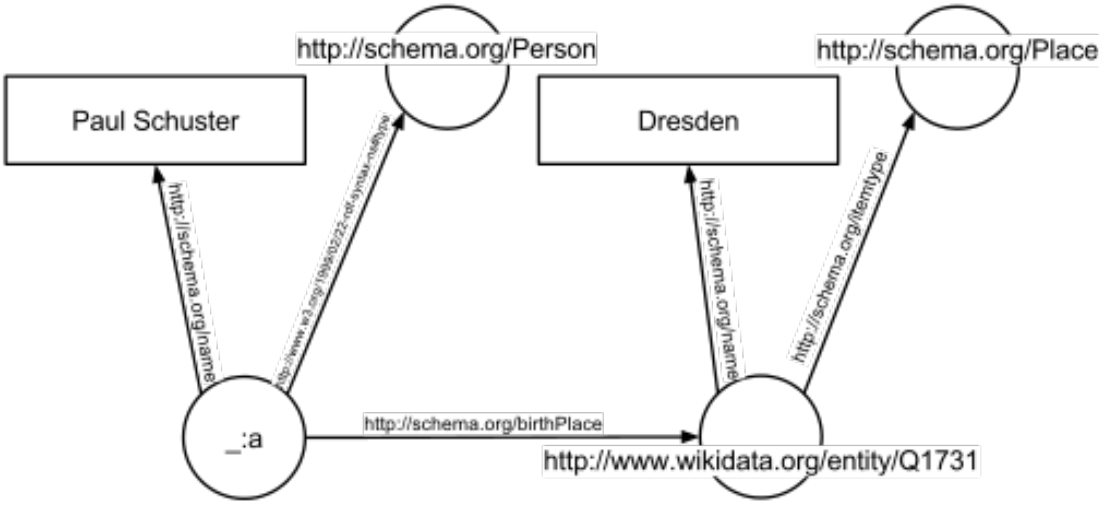


Figure 4.4: An example of semantic web

The node here called "_:a" is a node of type "person", a URI (`http://schema.org/Person`) defines what a "person" is. The "person" here has two different properties: "name" and "birthPlace". The "name" property is a string ("Paul Schuster"), a "terminal" (because it is not expanded, identified by a rectangular shape) property of the category "person". The property "birthPlace", on the other hand, is not a "terminal one", identified by a circle shape. It is a "place",

another web identity. For this reason it has its own definition with another URI (http://schema.org/Place) and it can have others properties (here not reported) to link a "place" to the remaining rest of the web.

Having given a first explanation of what linked data and semantic web are, we can now proceed to explain JSON-LD and its features useful for my purpose. JSON-LD is a type of document, coming from the JSON, that keeps its structure and its syntax, explained in 4.3.2, but it adds some keywords in order to be part of the semantic web, and to better describe a smart space. In particular the JSON-LD document uses the concept of *@context* as starting point. The @context can be explained as a context of a normal conversation: speaking with someone I have some external elements that help the interlocutors to understand each other, the environment, the place, the weather, etc. The concept is the same, trying to contextualize a conversation between two machines. Here the context maps the *IRIs (Internationalized Resource Identifier)* (more general URIs, using Unicode instead of ASCII) to *terms* defined in the document. Terms are case sensitive and any valid string that is not a reserved JSON-LD keyword can be used as a term. Then, in the rest of the document, we can refer to a particular definition given by the IRI, such as "place", with the term, without the need to map it every time. A @context can be defined internally, mapping explicitly every term to an IRI or can be defined online somewhere, in this case it is considered external.
The new format does not bring only the @context keyword: some others keywords are fundamental to make the web a semantic graph for a computer. The tag *@id* is introduced to correctly identify nodes inside the web, the @id component contains the place, in form of IRI, where the representation of the resource, and in particular of the node, is. The @id tag can be considered the "arrow" the parser has to follow in order to retrieve the node of the graph. @id is not pointing to a definition of what the resource is but the real representation of it.

Listing 4.4: JSON-LD: @context, @id example

```
{
  "@context": {
    "name": "http://schema.org/name",
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
  }
}
```

Reading the above listing 4.4 it is important to point out that in the @context two terms have been defined, "name" and "homepage", in particular the latter one uses the @id tag to point to the resource and @type that is explained in the next paragraph.
Another important tag, as already anticipated above, is *@type*, it has a double meaning: it can define the type of the node, it usually points to an IRI containing the formal description of a type of node. The other use of the @type tag is to define the type of a value: in this case it is used with the tag *@value* outside the

@context, or inside it defining a term without the need of @value. What is bind to a value type can be a type defined externally, always a IRI helps doing that, or a native JSON type: in this case there is no need of an external link, the JSON parser already knows how to handle it.
Type coercion is another feature which deserves to be explained: it allows someone deploying JSON-LD to coerce the incoming or outgoing values to the proper data type based on a mapping of data type IRIs to terms. Using type coercion, value representation is preserved without requiring the data type to be specified with each piece of data. Type coercion is specified within an expanded term definition using the @type key.

Listing 4.5: JSON-LD: @type for node and value example

```
1 "@context" : {
2   "@id": "http://example.org/posts#TripToWestVirginia",
3   "@type": "http://schema.org/BlogPosting",
4   "modified":
5   {
6     "@value": "2010-05-29T14:17:39+02:00",
7     "@type": "http://www.w3.org/2001/XMLSchema#dateTime"
8   }
9 }
```

In the listing 4.5, at line 3 we have a node type, while at line 7 we have a value type: the term "modified" is a of type "dateTime" while the node is of type "BlogPosting".
Finally, I would like to explain the concept of *active context*: the @context in a JSON-LD document is not unique, there is the possibility to define more than one, mapping new terms or reusing some already defined. The active context is how the JSON processor handles the presence of multiple context: a list of terms is kept by the processor, updating the IRIs mapped in case of redefinition. So redefining a term overwrites the old "value", keeping active only the last value. Between the two definitions of the same term, obviously, the first value is the one that is active. Each @context in a multiple context definition is called *local context*. Setting a local context to "null" resets the entire active context.

Listing 4.6: JSON-LD: multiple @context example

```
1 {
2   "@context": {
3     "name": "http://example.com/person#name",
4     "details": "http://example.com/person#details"
5   },
6   "name": "Markus Lanthaler",
7
8   "details": {
9     "@context":
10    {
11      "name": "http://example.com/organization#name"
12    },
```

```
13      "name": "Graz University of Technology"
14    }
15  }
```

In the example above 4.6, the term "name" is overridden in the more deeply nested "details" structure. Note that this is rarely a good authoring practice and is typically used when working with legacy applications that depend on a specific structure of the JSON object. If a term is redefined within a context, all previous rules associated with the previous definition are removed.
These explanation of JSON-LD is partial and it covers only the base aspects of it. [**jsonlddraft**] But it is sufficient to cover what I have used in my work.

4.3.4 W3C proposal: pros and cons

The W3C, during the years, has already faced the problem of lack of standards in the highest layer of the structure. So it has published an unofficial draft, from which the idea of the thesis has been taken, discussed and changed in some parts, according to my personal needs and the thoughts, discussed and accepted by the professor [**w3cwot**].
In particular I took from the W3C idea two main things: firstly the idea of a REST service, a "consumable" solution. In particular I found the idea of a "semantic solution", in order to use the full potential of linked data, very innovative and looking to the future. Secondly I shared the "thing description" the W3C proposes: in particular it divides the capabilities of a smart object into three different interaction patterns: *properties*, *actions* and *events*.

- Property: it provides readable and/or writeable data that can be static (e.g., supported mode, rated output voltage, etc.) or dynamic (e.g., current fill level of water, minimum recorded temperature, etc.).
- Action: it targets changes or processes on a thing that take a certain time to complete (i.e., actions cannot be applied instantaneously like property writes). Examples include an LED fade in, moving a robot, brewing a cup of coffee, etc.
- Event: it enables a mechanism to be notified by a thing on a certain condition.

It is fundamental to point out that I am not implementing what they are proposing without making reflections, changes and thoughts to the draft. W3C is nowadays an essential component to standardize protocols and ways of communication on the web, it is composed of web experts having years of experience; it can be considered "somehow" natural that an idea for a thesis comes from someone or somewhat having more influence on the existing technologies than a single student.
Having specified that my work will anyway introduce something new respect to the draft now I am going to list briefly the innovations compared to W3C ideas. Innovations, more generally, are then explained and presented in the following

sections and chapters, without making a schematic comparison continuously, because they are only a part of a bigger and heterogenous work.
Firstly, my idea is provide the developer an indirectly the final user with a description of the whole smart space: a smart space is a very heterogeneous term, it can be outdoor or indoor, composed by different zones or rooms, etc. So my idea is to face all these categories to create interchangeable blocks of my JSON-LD document. In each of these zones, maybe, the same type of device is installed, so the problem to uniquely identify a smart object is obviously one of the fundamental requirements and challenges to face. Also the discovery of devices can be something to care about: the object in the right place is another challenge to be overcome to have a full working system. Secondly, W3C has not specified "how to use" what they are proposing: I decided to put the entire system on local host for some cases and online for others, recommending the developers that will work on the control centers to do the same. Even if there must be some exceptions. The advantages of putting the "smart world" on the web, creating the WoT, have already been explained and presented. The advantage of putting the system on local host is the fact that access is restricted without using any password, it can be useful for example for "public" smart spaces: being in a hotel room I must have access to all the devices in my room but not to others or to the centralized system in the hotel. So a customer can enter the room and control everything without need of authentication. On the other side the owner of the hotel needs to have a complete vision of what is happening in his/her structure: in case of an emergency it is necessary to control everything remotely, assuring that who is using the system has the permissions to do it. This simple case ranges among all the shades the problem can have. Maybe other minor features and changes to the problem are here not listed but the whole work is explained in the next sections.

4.3.5 Syntax of the solution

In this section I am going to define step by step my solution, explaining with listings and images all the elements.
An important and remarkable reflection is needed here: unfortunately it is not possible for me alone to define every kind of smart space and smart objects existing nowadays. I will provide the reader a sample of what I am doing and what is the work. The examples are obviously made to cover all the different shadows of the matter, in order to give a complete idea and a full functional system.

Context Here I would like to explain the keyword @context and its meaning in my work. As said, @context must define the terms are used in the document, so they change for each particular smart space considered. Here I am reporting an example, that could maybe change on the proof of concept

Smartspace

Chapter 5

Proof of Concept

5.1 Non-functional requirements

If the section 3.3, the constraints of the problem can be seen as the functional requirements my development has to have, there is another category of requirements: the non-functional ones, they are important properties that my system must have in order to guarantee full functionalities. They are not specific for the Web of Things but it is important that my system meets them. I am briefly here listing all the non-functional properties:

- *Portability:* to have my application users by the largest number of users possible, but this is not a real requirement in the sense that the application is developed on the web, easy reachable. We are on the border of functional and non-functional requirements.
- *Stability:* system must be always available, and able to offer all its services. For example I should avoid possible system failures during the automatic reload of the configuration in case of a change in the smart space. In addition, data must be durable and not lost for any reasons.
- *Availability:* the services must be always accessible in time. In case of malfunctioning, administrator will provide maintenance in order not to affect service availability.
- *Reliability:* since data are shared among a large number of devices, reliability is essential. Users can base their actions on other users' actions and on devices' status. Moreover, I suppose that the memory where data are stored is stable.
- *Efficiency:* Within software development framework, efficiency means to use as less resources as possible. Thus, system will provide data structures and algorithms aimed to maximize efficiency. I will also try to use well known patterns reusing as many pieces of code as possible, taking care of avoiding any anti-pattern.
- *Extensibility:* My application must provide a design where future updates are possible. It will be developed in a way such that the addition of new

functionalities will not require strong changes to the internal structure and data flow.

- *Maintainability:* Also modifications to code that already exists have to be taken into account. For this reason the code must be easily readable and fully commented.
- *Security:* Using an online service security is always required. The fact that the system will be available only on LANs is a one first step in this direction.

5.2 Package organization

probabilmente diventerà subsection, solo citata qua per non dimenticarla + test cases, cost estimation? (cocomo etc)

Chapter 6

Conclusions and Future Works

Figures Copyright

Chapter 2: State of Art

- Figures **??**, **??**, **??**, **??** and **??** are taken from [**guinard2011web**] and reproduced with the kind authorization of the author, citing his personal website http://dom.guinard.org.
- Figures **??** and **??** are reproduced with the kind authorization of DEIB, my department in Politecnico di Milano and its professors.
- Figure **??** is a composition of two different images. Left part is subject to a public domain license. The license notice is available at https://commons.wikimedia.org/wiki/File:Ipv4_address.svg, which allows the reuse. Right part is subject to a public domain license. The license notice is available at https://commons.wikimedia.org/wiki/File:Ipv6_address.svg, which allows the reuse.
- Figure **??** is taken and reproduced with the kind authorization of OpenPicus, www.openpicus.com.

Chapter 4: Solution

- Figure 4.1 are taken from the Apple WWDC 2014 informative pdf http://devstreaming.apple.com/videos/wwdc/2014/701xx8n8ca3aq4j/701/701_designing_accessories_for_ios_and_os_x.pdf, and according to http://www.apple.com/legal/internet-services/terms/site.html "Content" section, there is the possibility to reproduce the content, following the 4 points there listed.
- Figure 4.4 is subject to a public domain license. The license notice is available at https://commons.wikimedia.org/wiki/File:RDF_example.svg, which allows the reuse.

The images that are not explicitly listed are made by me and no copyright is needed.