



University of Padua

DEPARTMENT OF INFORMATION ENGINEERING
Master Degree in Computer Engineering

MASTER DEGREE THESIS

Orchestration of smart objects with MQTT for the Internet of Things

Candidate:
Gianmarco Nalin
ID Number: 1033984 - IF

Thesis advisor:
Ch.mo Prof. Carlo Ferrari
Research supervisor:
Ing. Michele Stecca

Academic Year 2013 - 2014

"The only way to do great work is to love what you do."
Steve Jobs

To Elena, the strongest woman I have ever met.

Abstract

At the beginning of Internet, users were only consumers of information, in what it is called **Web 1.0**. Thanks to the integration with *databases* and *content management systems* (CMS), the way users interact with the Web has changed. People can become also creator of contents and, therefore, contribute to the emerging **Web 2.0**.

In this scenario, many companies make available several web services to interact with, permitting users and developers to create custom applications.

At the same time, devices permit users to remain connected everywhere and anytime. Not only smartphones, tablets and PCs but also fridges, washing machines and cars. These new connected devices can exchange information with the real world in order to remotely act with them. This is the **Internet of Things** (IoT), where objects and services coexist e cooperate.

The IoT is evolving as fast as companies and developers identify new opportunities of growth. This growth is possible if there exist protocols which enable *interoperability* between different devices and networks. In this direction, IBM has developed MQTT, a new protocol which facilitates devices integration and management in constrained environment.

In this thesis, we will describe a platform which permits the creation of **composite services** (a.k.a. mashups) that combine several web services and physical devices to execute custom tasks.

In Chapter 1, we will introduce the Internet of Things describing some nowadays architectures. In Chapter 2 we will give a short description of the iCore project, which is the work's starting point. In Chapter 3, we will describe the publish/subscribe paradigm as well as the MQTT protocol. In Chapter 4, we will introduce a creation and execution composite services platform. Finally, we will state results and future developments about the project.

Contents

1	An introduction to the Internet of Things	1
1.1	Semantic meaning of Internet of Things	1
1.2	Main Challenges for the IoT	2
1.3	Constrained Application Protocol and Machine to Machine Communications	2
1.4	IoT applications	3
1.5	The Web of Things	4
1.5.1	RESTful Architectures	5
1.6	Case study: Xively [®]	7
1.6.1	How it works	9
1.6.2	Pricing	11
2	The iCore Project	13
2.1	Service Creation Platform	13
2.2	Service Execution Platform	14
2.2.1	Service Proxy	15
2.2.2	Orchestrator	15
3	A publish-subscribe protocol: MQTT	19
3.1	Publish-Subscribe Systems	20
3.1.1	The programming model	21
3.2	Message Queue Telemetry Transport	22
3.2.1	Message format	23
3.2.2	Message flows	25
3.2.3	MQTT Applications	27
3.3	An MQTT implementation: Paho	29
3.4	Terracotta [®] Universal Messaging: setting up a local environment	29
3.4.1	Terracotta Universal Messaging	30
3.4.2	Client Implementation with Paho MQTT	34
4	Integrating MQTT into the platform	37
4.1	Centralized SEP	38
4.1.1	Example	38
4.2	Distributed SEP	44
4.2.1	Example	45
4.3	Centralized and distributed implementation comparison	51
	Conclusion and future developments	53

A Publisher Java Code	55
B Subscriber Java Code	57
Bibliography	59

Chapter 1

An introduction to the Internet of Things

Nowadays, we have the ability to measure, sense and monitor everything in the physical world. How can we use these information to increase business productivity, improve human health and monitor environment situation? Can we make them publicly available in order to enhance the knowledge? A Cisco study [12] states that in 2020 there will be about **50 billion** of connected physical world devices, fueled by a $1000\times$ increase in wireless broadband traffic. This is a huge number of devices and their interoperability is a challenge these days: we need protocols that make possible the communication between all these devices.

In the following sections, we will present an overview about what we intend with Internet of Things (IoT) using practical examples followed by a case study of a company whose core business is founded on IoT.

1.1 Semantic meaning of Internet of Things

Huang and Li in [2] analyze the semantic meaning of *Internet of things*: generally, the nouns in phrase like “*noun1 + of + noun2*” are related in some way and have an affiliation relation or an apposition relation. With the phrase *Internet of things*, we have an exception: the relationship between the two nouns is not the one described above but a more exact understanding is “*the Internet related to things*”.

The main function of Internet is to interconnect, using cables, optical fiber or microwaves, the computer terminals all around the world. The objects transmitted in form of electric or optical signals cannot be *things* as material entities but only the information as immaterial ones. So, the semantic meaning of *Internet of things* is the “Internet relating to information of things”, where with the term “relating to” we intend that the information produced by things flows rationally on the Internet in order to be shared all around the world.

1.2 Main Challenges for the IoT

There are several challenges to which the IoT vision should overcome: contextual (including policy) and technical applications are the main ones. In a world where everything is interconnected, where data about local environment (and about humans in direct or indirect way) are exchanged, **privacy** is fundamental and it has to be guaranteed and protected. The individual's trust to the IoT should be complete and information about negative impact on individual or society has to be safeguarded.

Standardization of technologies is also important because it will lead to better interoperability with a reduction of the entry barriers. Nowadays, each manufacturer produces and implements its own solution based on vertical integration of the entire system. The change from *Intranet of Things* to a more complete *Internet of Things* is a fundamental requirement.

So, the key challenges areas are:

- **Privacy, identity management, security and access control:** who can share and see with which credentials is a significant challenge.
- **Standardization and interoperability:** how can we guarantee that all the technology platforms continue to act, as a unique and coherent platform, where we do not to re-invent the wheel every time we have to develop a new application or we have to add a new sensor to the system?

1.3 Constrained Application Protocol and Machine to Machine Communications

The term **machine to machine** (M2M) refers to technologies that allow wired and wireless systems to communicate with other devices of the same type. M2M does not refer to specific network, information and communication technologies. It is particularly useful for business executives.

Modern M2M communication has expanded beyond the one-to-one connection and changed into a systems of networks that transmit data to personal appliances. With the world adoption and expansion of IP networks, M2M communication has begun to take place with the reduction of the amount of power and time necessary for information to be collected and transmitted between machines. A possible protocol that allows M2M communication is the Constrained Application Protocol.

Constrained Application Protocol (CoAP) is an application level (ISO/OSI level 7) protocol which is suitable for very small and resource-limited devices. The protocol stack is showed in Figure 1.1. It allows devices to communicate interactively over the Internet. CoAP is ideal for small low power sensors, switches and similar devices which have to be controlled over the standard Internet network.

CoAP is designed to be a **RESTful** protocol that permits both **synchronous** and **asynchronous** communication. It is specialized for **Machine to Machine** and **IoT** applications and it permits an easy proxying to/from HTTP. The protocol is not designed neither to be a replacement for HTTP nor deeply separate from the Web, but to be extremely merge with it.

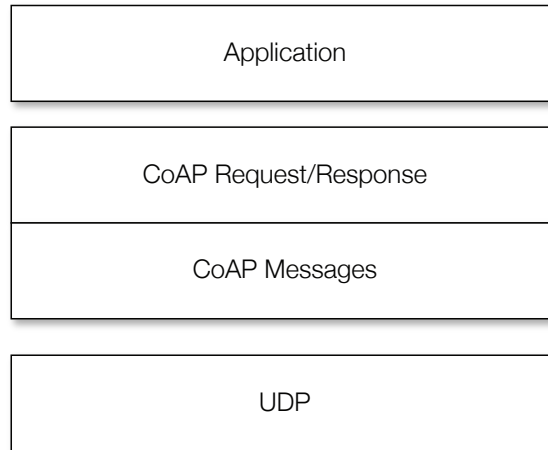


Figure 1.1: CoAP position in the ISO/OSI architecture.

1.4 IoT applications

In this section, we present some of the possible applications of the Internet of Things. As in [1], we can group everyday situations in several sets, showing how IoT is becoming a technological revolution.

- SMART CITIES

Smart parking Monitoring of parking spaces available in the city;

Smart health Monitoring of health status of buildings, bridges or historical monuments;

Noise urban maps Sound monitoring in central zone in real time

Traffic congestion Monitoring of vehicles and pedestrian levels and optimize routes;

Waste management Detection of rubbish levels in containers and optimize trash collection routes.

- SMART ENVIRONMENT

Forest fire detection Monitoring of combustion gases and fire over critical zones;

Air pollution Monitoring and control of CO₂ emissions of factories and farms.

- RETAIL

Supply chain control Monitoring of stock conditions and product tracking;

Near Field Communication (NFC) payments Payment processing for public transport, gyms, coffee shops, etc.

- INDUSTRIAL CONTROL

Machine to machine (M2M) applications Machines auto-diagnostic and assets control.

- DOMOTIC AND HOME AUTOMATION

Energy and water use Monitoring and control of water and energy usage in order to save money and resources;

Remote control appliances Switching on and off remotely appliances in order to avoid accidents.

- EHEALTH

Patients surveillance Monitoring and control patient condition in his hospital room or in his own home;

Fall detection Assistance for older or disabled people to make them living independent.

1.5 The Web of Things

The creation of *smart things* and their interconnection through a network has become a goal of many research activities. Rather than creating a brand new vertical architecture specific for this purpose, it has been proposed to make these objects an integral part of the Web.

In this scenario, popular Web technologies such as **HTML**, **JavaScript**, **Ajax**, **PHP**, **ASP.NET**, can be used to build applications that involve *smart things* and users can leverage well-known Web mechanisms, e.g. browsing, searching, bookmarking, linking, to interact with and share these devices.

A first proposal came in [5] by Kindberg et al.: they proposed to link physical objects with Web pages which contain information and services. By the use of infrared reader or bar codes on objects, users can retrieve the URI of the associated page with a simple interaction paradigm.

Another interaction model has been provided in [6] by Guinard et al. in which they propose to incorporate real world smart objects into a standardized Web service architecture, e.g. using **SOAP** (Simple Object Access Protocol), **WSDL** (Web Services Description Language), **UDDI** (Universal Description Discovery and Integration). In practice, such integration method is too heavy and complex for objects with limited resources.

Recently, several "Web of Things" projects have explored simple embedded HTTP servers and **Web 2.0** technologies. Thanks to TCP/HTTP cross-layer optimisations, web servers, with advanced features (e.g. concurrent connections or server push for event notification), can be implemented with only 8KB of memory without the OS support. In this concept, smart things and their services are completely integrated in the Web by reusing and adapting technologies used for traditional Web contents. **REST** architecture well integrates and completes this point of view providing a simple uniform interface and mechanisms for clients to choose the best possible representations for interactions. In the next section, we provide a brief introduction to REST architectures with some examples.

1.5.1 RESTful Architectures

Introduction

REpresentational State Transfer (REST) is a type of software architecture for distributed hypermedia systems. The term was introduced in 2000 by Roy Fielding in his doctoral thesis. REST-style architectures conventionally consist of *clients* and *servers*. Clients create requests to servers; servers process requests and get back the appropriate responses. Requests and responses are built around the transfer of resources representations. A **resource** can be any coherent and meaningful concept that may be addressed. A **representation** of a resource is a *document* that capture the current or intended state of that resource.



The REST architectural style has six main constraints about architecture while it leaves implementations of individual components free to design.

Client-Server: a uniform interface separates clients and servers. This separation means that, for instance, clients don't have to worry about data storage, which remains internal to the servers (this improves the client code portability); from the servers' point of view, this concept means that servers don't have to worry about user interface or user state (this makes the server simpler and more scalable). Clients and servers may be replaced and developed independently as long as the interface between them is not modified.

Stateless: communications between clients and server have to be **stateless** such that each request must contain all necessary information to understand it and it cannot take advantage of any stored content on the server. Session state is kept entirely on the client.

Cacheable: in order to improve network efficiency responses to requests may be client cached. Servers can explicitly label responses as *cacheable* or *non-cacheable*.

Layered systems: the layered system style allows an architecture to be composed of several hierarchical layers by constraining component behavior such that each component can see only the public interface of the component it is interacting with without knowing the specific implementation.

Code on Demand: REST allows client functionality to be extended by downloading and executing code in form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented and allowing features to be downloaded after the system deployment.

Uniform interface: the interface between clients and servers simplifies and separates the architectures, giving to each part the ability to evolve independently.

RESTful Web APIs

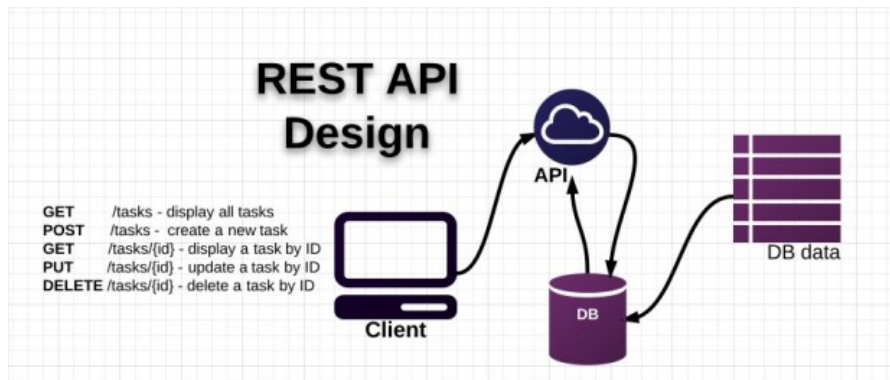


Figure 1.2: REST API Design.

A RESTful web API (also called a RESTful web service) is a web API developed using HTTP and REST principles. As you can see in Figure 1.2, it is a collection of resources with defined aspects:

- the base URI for the web API, such as `http://firm.com/employees/`;
- the *Internet media type* of the data supported by the web API: this is often **JSON** but it could be any valid Internet media type;
- the set of operations supported by the web API using HTTP methods (e.g. GET, POST, PUT, DELETE);
- the API must be hypertext driven.

In the following table we summarize the recommended return values of the primary HTTP methods in relation with the resource URIs.

Interface Guidelines

The uniform interface constrain described in the previous section is consider fundamental to the design process of any REST service.

Identification of Resources: each resource is identified in requests, for instance using URI in web-based REST systems. Also the resources are conceptually separated from their representations that are get back to the client; usual representations are HTML, XML, JSON.

HTTP Verb	Entire Collection e.g. /employees/	Specific Item e.g. /employees/id/
GET	200 (OK) - list of employees.	200 (OK) - single employee. 404 (Not Found) if ID is not found or invalid.
PUT	404 (Not Found), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content) - update employee information. 404 (Not Found) if ID is not found or invalid.
POST	201 (Created) - Location header with link to /employees/id/ containing the new ID	404 (Not Found) - Generally not used.
DELETE	404 (Not Found), unless you want to delete the whole collection - not often desirable.	200 (OK). 404 (Not Found), if ID is not found or invalid.

Manipulation of resources through these representations: when a client holds a representation of a resource, including any attached metadata, it has enough information in order to modify or delete the resource on the server, if it has permissions to do that.

Self-descriptive messages: Each message has enough information to describe how to process it. In addition, responses explicitly indicate their cacheability.

1.6 Case study: Xively[®]

Xively is a company already known as *Pachube* first, and *Cosm* later, which focuses its business in building a **platform** for the IoT.

It offers their services as a **Platform as a Service** (PaaS): it provides a web environment where it is easy to develop and deploy custom applications that use the Internet of Things.

It is also available a set of libraries (Java, PHP, C, Javascript, Objective-C and many others) that make easy Xively integration on a very large number of devices and fields.

In Figure 1.3, we show the Xively platform overview.

The provided services are conceptually simple:

Directory Services They make simple the indexing and searching of objects and permissions on a possible very large number of devices and users.

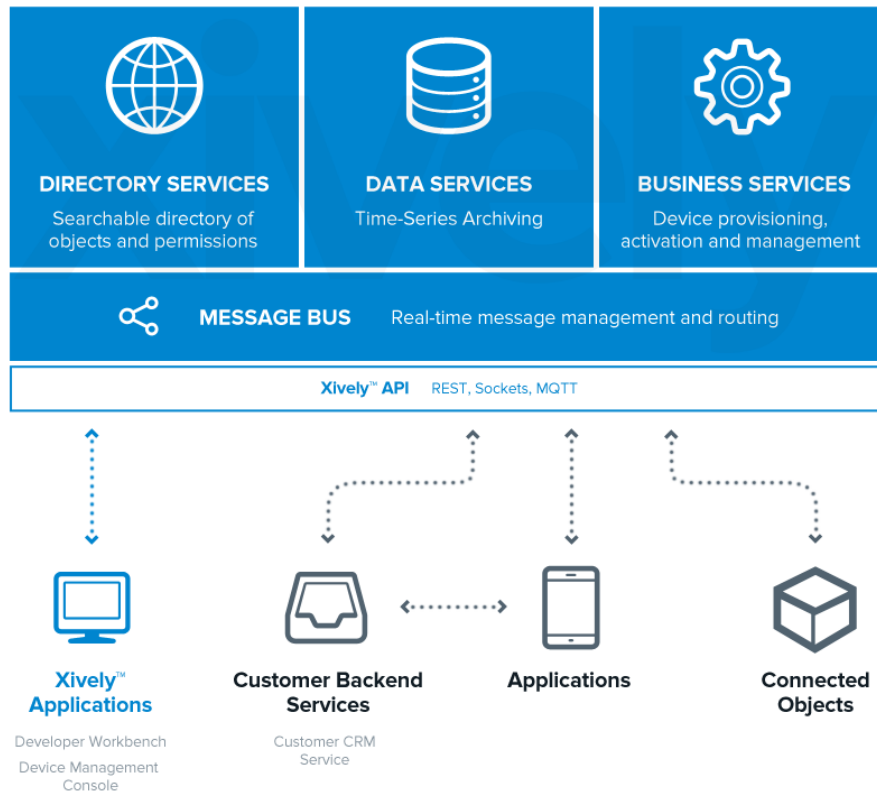


Figure 1.3: Xively platform overview.

Data Services They store data from the devices in order to get to users the possibility to compute statistics on a time basis.

Business Services They simplify the work that has to be done in order to deploy, register, activate and manage every single device.

Everything is hidden by an API level that makes easier the development of applications and software that use Xively.

The key concepts are:

Product It is the highest level of abstraction in Xively: it describes the common information of the same type of devices.

Device It describes a single and unique device. It is identified by an ID and every data are exchange using the Feed

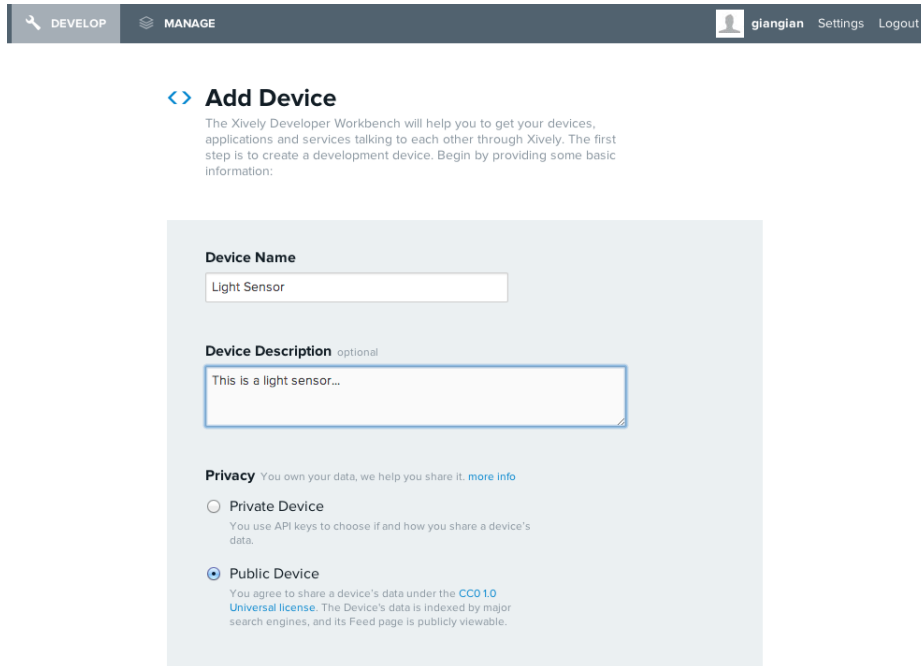
Feed It is the stream of information associated with every single device. It is identified by a unique ID.

Channel It is the source of a single type of information. A channel could be the temperature sensor of a device. Multiple channels could be associated with a single device.

1.6.1 How it works

In order to better understand how Xively works, we propose a practical example. We used a developer account which has some limitations on the usage.

The first thing to do is adding a new development device. In Figure 1.4, we show the screenshot of this procedure.



<> Add Device

The Xively Developer Workbench will help you to get your devices, applications and services talking to each other through Xively. The first step is to create a development device. Begin by providing some basic information:

Device Name

Light Sensor

Device Description optional

This is a light sensor...

Privacy You own your data, we help you share it. [more info](#)

Private Device
You use API keys to choose if and how you share a device's data.

Public Device
You agree to share a device's data under the [CC0 1.0 Universal license](#). The Device's data is indexed by major search engines, and its Feed page is publicly viewable.

Figure 1.4: Xively platform: adding a new development device.

Once confirmed every information, we are brought to the management screen as showed in Figure 1.5. In this section you can retrieve the **Feed ID** associated with the device, the **Activation Code** for activate the device, add new **API keys** and specific **Channels** for the device. As you can see in the same figure, when a new device is created, Xively creates also an **auto-generated** API key with the permission to do everything: you can generate other keys in order to create a suitable permission scheme for the device.

The section presented above is dedicated to the development and debug of device's application. Once the application is mature and bug-free, you can easily deploy the device in production environment. What if you have hundreds of temperature sensors, should you add every single sensors one by one? The answer is obviously no: Xively permits to insert a **batch** of devices, uploading a CSV file which contains all the devices' serial numbers. This procedure simplifies enormously the initial activity of adding a lot of devices. This feature is showed in Figure 1.6.

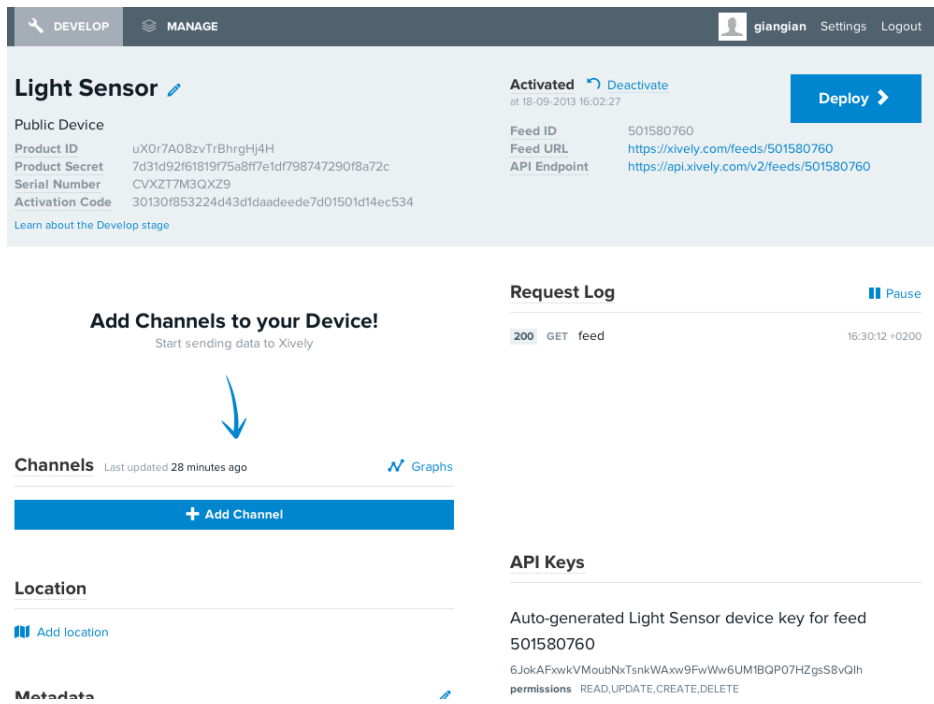


Figure 1.5: Xively platform: device's resume screen.

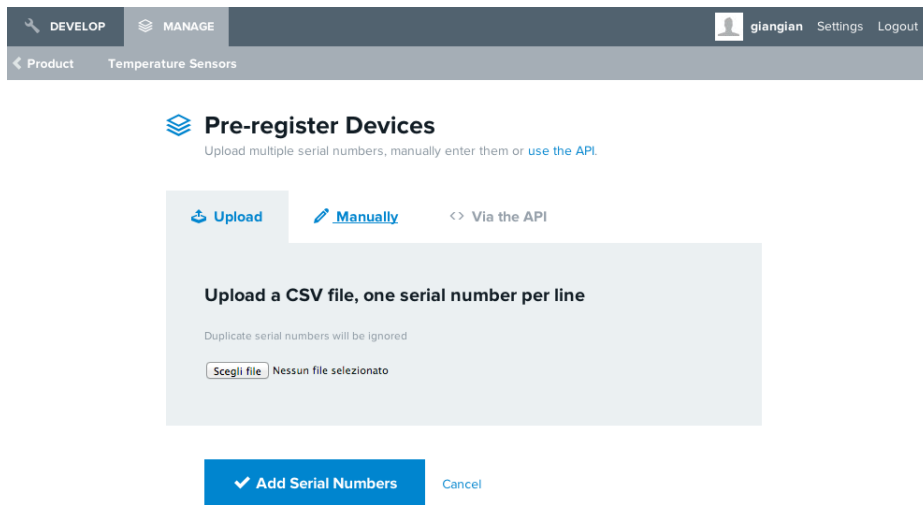


Figure 1.6: Xively platform: batch adding.

1.6.2 Pricing

Xively gives to users three types of service, from an entry-level service to a top one. The main differences among these three types consists in the hours of available consulting support and the price of channels per transaction volume. The service pricing table is showed in Table 1.1.

	Starter	Value	Select
Priority Support	12 hours	12 hours	12 hours
Consulting Services	6 hours	20 hours	48 hours
Channel Pricing	Low: \$1.25 Average: \$2.00 High: \$3.25	Low: \$0.99 Average: \$1.55 High: \$2.50	Low: \$0.65 Average: \$0.99 High: \$1.25
Price	999\$/year	4,900\$/year	39,000\$/year

Table 1.1: Xively service pricing table.

Chapter 2

The iCore Project

iCore is a EU funded project which has two main objectives:

- **abstracting** the technological heterogeneity which comes from the huge amount of heterogeneous objects, while maintaining the *reliability*;
- considering the views of different users and stakeholders in order to ensure proper application provision, business integrity and **exploiting business opportunities**.

In order to run a mashup, we need a **platform** in which users can deploy these composite services and execute them. Stecca and Maresca in [15] proposed a platform, based on a **Request/Response** model, that supports the execution of **server-side/event-drive** mashups. In the following sections, we will describe the two main components of this platform, the **service creation platform** (SCP) and the **service execution platform** (SEP): the former permits the creation of mashups and the latter permits their execution.

2.1 Service Creation Platform

The creation of a mashup is made through a **graphical interface** in which creators can drag and drop **blocks** representing services and define the relationships between them by drawing an **edge** from the emitting service to the receiving one, as shown in Figure 3.1. Once the mashup is created, it is saved as an XML file and stored in a repository. An example of the editor is shown in Figure 2.1.

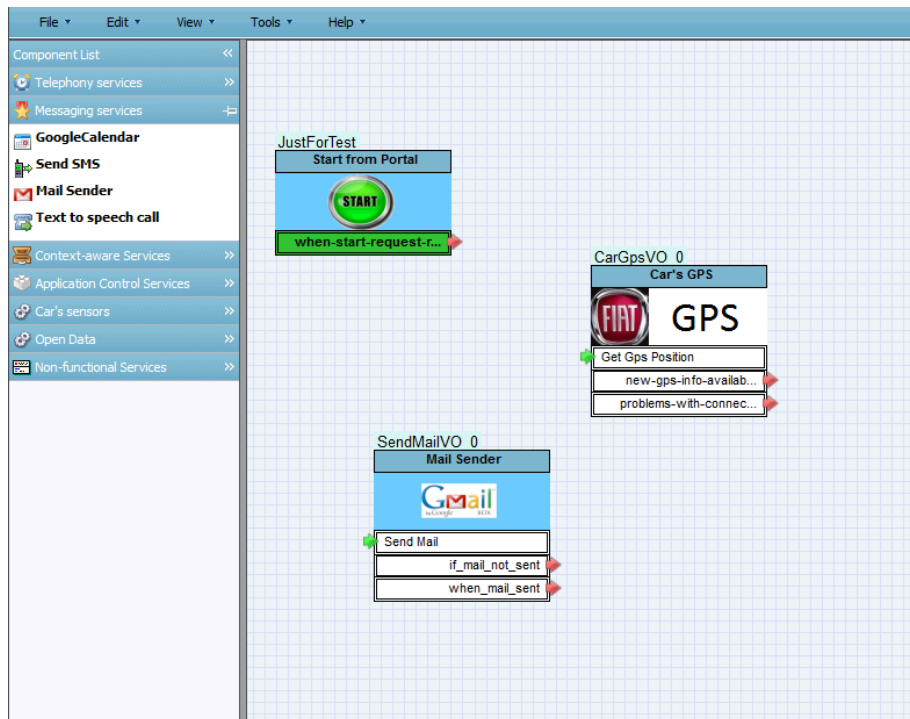


Figure 2.1: Graphical user interface for creating mashups.

2.2 Service Execution Platform

The service execution platform is the actual environment in which mashups live and compute. It has to satisfy a set of requirements, such as:

- **Scalability:** the system has to support the simultaneous execution of a possible large number of instances (also called *sessions*) on different hardware size;
- **Fault Tolerance:** the system has to ensure that even if some components fail, the session is able to continue its execution;
- **Low latency and high throughput:** The system has to serve as many requests as possible, according to the hardware characteristics;
- **Authentication, Authorization and Accounting (AAA):** the system has to guarantee the opportunity of managing users' permissions, different service levels (SL) and authorization;
- **Management:** the system has to permit a full control over the available resources in order to allow to the platform administrator the application of necessary maintenance operations (e.g. enforcing *Service Level Agreement* rules).

2.2.1 Service Proxy

Although the existing services use standard interfaces (e.g. RESTful API, SOAP) to expose their resources, they do not expose interfaces compatible with event-driven platform because of the different technologies and different data formats (e.g. XML, JSON, YAML). In order to ensure the compatibility with the system, Stecca and Maresca introduced a layer between the internal system and the external world. They defined the concept of **Service Proxy** (SP): an SP wraps an external resource and makes it compliant with the event driven model previously explained. Given this concept, a different SP is needed for each external resource we want to make available in this platform. SP translates input properties into specific external service format and parses the service response to fulfill the output properties, as shown in Figure 2.2.

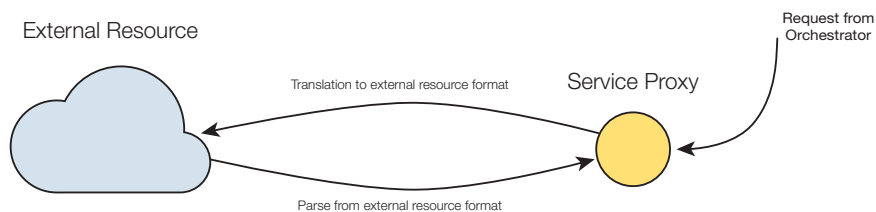


Figure 2.2: How a SP interacts with external world.

In order to receive events from the orchestrator node (ON), SPs expose a primitive called `invokeAction` through which the ON activate the SP functionality. Within a SP, we can identify two types of communication:

- **Mashups level**, which refers to the `invokeAction/notifyEvent` calls in relation to the mashup logic;
- **External resource level**, which refers to the communication between the wrapper SP and the external service.

As we said before, only one SP is needed for each external resources although several instance of the same SP can be deploy for performance and fault tolerance reasons.

2.2.2 Orchestrator

The component that actually executes mashups according to defined constrains is called **orchestrator**. This software executes mashups logic combining services as defined in the SCP. As we said in the previous section, the orchestrator does not communicate directly to the external services but it receives and forwards events from and to SPs inside the Service Provider Domain (SPD). Figure 2.3 depicts the platform architecture and how the orchestrator works.

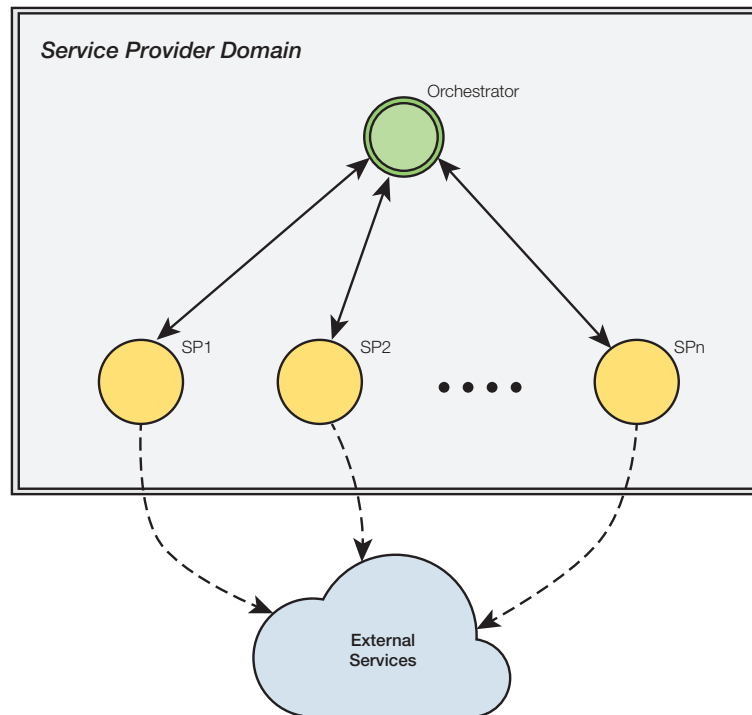


Figure 2.3: High level platform architecture.

In the platform, a single ON is needed but, for performance reasons, several instances may coexist at the same time. When a new deployment request arrives, each active ON retrieves from repository the XML representation of the requested mashup. According to the XML, ON translates it to a **Routing Table** that is used for managing the mashup logic during its execution. Each table entry is identified by the pair $\langle \text{senderSPId}, \text{eventId} \rangle$, through which the ON can get the information required for invocation of the next action or actions in the mashup.

Regarding a single action, the required information for the correct forwarding of the next action are:

- the identifier of the next action;
- the identifier of the next SP;
- the parameters assignment from output to input values.

Now we have a clear idea about which information is needed to correctly execute the mashup. Next step is to define the algorithm through which a composite service runs. When an event occurs on an ON, the steps to follow are described in Algorithm 1:

Algorithm 1 Next action(s) invocation algorithm.

```

1: NEXT ACTION(EventId, SenderSPId, SessionId, Properties)
2: MashupId  $\leftarrow$  SessionId.MashupId
3: RoutingTable  $\leftarrow$  getRoutingTable(MashupId)
4: entries  $\leftarrow$  RoutingTable.getEntries(<SenderSPId, EventId>)
5: for all entry  $\in$  entries do
6:   Update Properties based on assignment fields.
7:   entry.nextSP.invokeAction(ONEndPoint, ActionId, TargetSPId,
   SessionId, UpdateProperties)
8: end for

```

As we can see in Algorithm 1, invoking next action, ON includes its endpoint as a parameter: this is due to the possibility that many ONs exist on the platform simultaneously and SP has to be able to notify the correct ON. Switching between ONs supports **load balancing**, **scalability** and **fault tolerance**.

Another important feature of orchestration mechanism is that no state information is stored during a mashup session: we refer to this property as **stateless orchestration**.

The mechanism listed in Algorithm 1 is depicted in Figure 2.4.

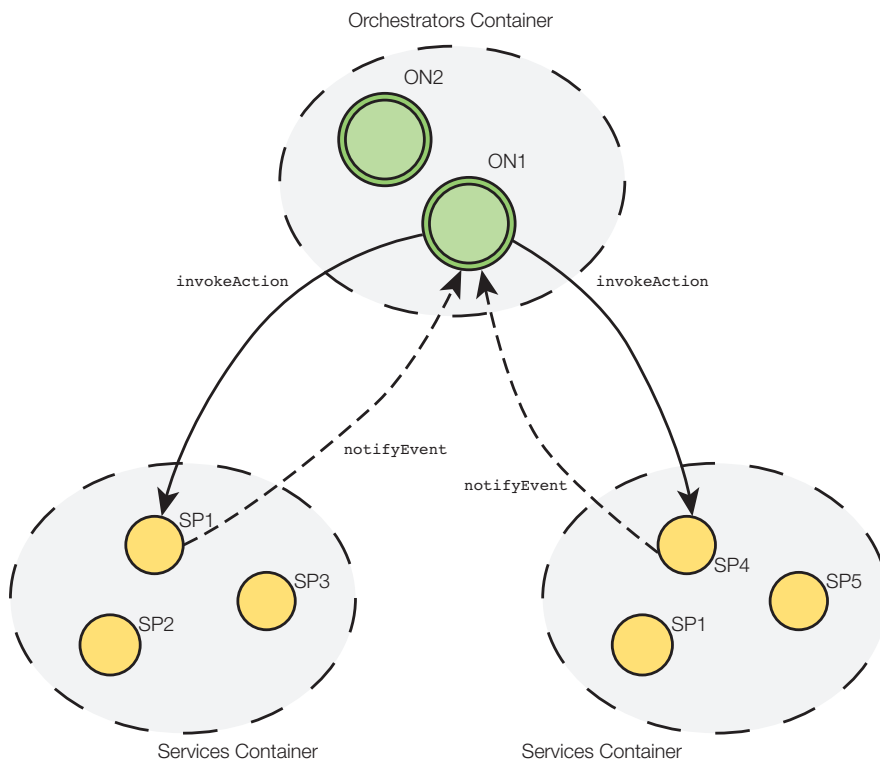


Figure 2.4: Run time execution model.

The platform described in this section is suitable for mashups of web services but with emerging of the Internet of Things, where thousands and thousands of devices and smart objects can be mixed with web services, this platform does not work as desired. The system architecture does not scale with the huge amount of new entities which want to be integrate to the platform.

The solution to this problem comes from the distributed system literature and it is called **publish-subscribe architecture**. These architectures guarantee **scaling** property and asynchronous communications which are fundamental in smart objects implementation and functionality (e.g. sensors).

The next chapter will introduce the publish-subscribe (P/S) architecture and propose a lightweight and P/S compliant protocol which is designed to be used in constrained applications such as the IoT ones.

Chapter 3

A publish-subscribe protocol: MQTT

The Internet gives a huge amount of free services which carry out different jobs, such as mail services, RSS feeds, Flickr[®] flows, etc. We can combine all these services into new custom *composite services* which enrich the benefits of every single service. We name this combination of services as a **mashup**. Figure 3.1 shows an example of mashup.

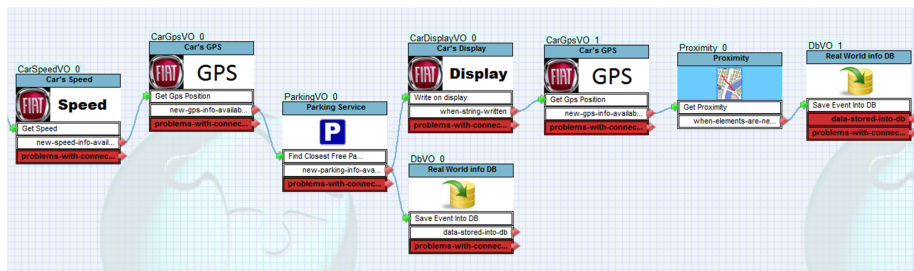


Figure 3.1: An example of mashup.

The idea behind this concept is really simple: each service exposes a set of **actions** it can execute and a set of **events** it can fire up. Using this paradigm, we can build chains of services which receive data from a service and produce information that can be used by another service. The above example has been built for a park searching assistant installed into a car. In this composite service we have six different services:

- **Speed:** the service reads the current car speed and fires an event when a new speed value is available;
- **GPS:** the service retrieves a GPS location and fires an event when a new location is available;
- **Parking service:** given a GPS location, the service gives the nearest free parking station from that location and fires an event when a parking station is available;

- **Database service:** given a set of information, the service saves these information in a database and fires an event when data are correctly saved into the database;
- **Display:** given a string, the service writes a string on the car's display and fires an event when the write process ends;
- **Proximity:** the service monitors the distance between a given GPS location and the current location and fires an event when this distance is below a defined threshold.

Platforms, which permits the creation of personalized mashups, already exist: they provide graphical tools in order to build up composite services without any particular programming skill. Such services includes **Yahoo! Pipes**¹, **JackBe Presto Server**² and **Apatar**³.

After a brief recall of publish-subscribe systems, we will introduce MQTT which are at the new platform fundamental components.

3.1 Publish-Subscribe Systems

The publish-subscribe paradigm is an **event-based** architecture in which *publishers* publish structured events to an event service (usually called *broker*) and *subscribers* show their interest in a particular event through *subscriptions*; these subscriptions can be custom patterns over the structured events.

These systems are used in several application domains, in particular those related to a *large-scale* events airing. Examples include:

- financial information systems;
- applications with live feeds of real-time data (including RSS feeds);
- support for *cooperative working* where users need to be informed of events of interest;
- support for *ubiquitous computing*, including the management of events emanating from ubiquitous infrastructure (i.e. location events);
- a broad set of monitoring applications, including *network monitoring* in the Internet.

This type of system is a key component of Google's infrastructure, i.e. in the dissemination of events related to *advertisements* [4].

Publish-Subscribe systems have two main characteristics:

Heterogeneity In a scenario where event notifications are used as a communication media, components in a distributed system that were not design for interoperability can be made to work together. The only required thing is that the event-generating objects publish the events they offer, and that other object can subscribe to event topics (i.e. communication channels) and provide a suitable interface for receiving and dealing with event notifications.

¹<http://pipes.yahoo.com>

²<http://www.jackbe.com>

³<http://www.apatarforge.org>

Asynchronicity Event notifications are sent asynchronously by publishers to all subscribers who are interested in them, in order to prevent the publishers needing to synchronize with subscribers. In other words, publishers and subscribers have to be *decoupled*.

3.1.1 The programming model

As shown in Figure 3.2, the publish-subscribe model consists of a small set of operations:

***publish*(*e*)** A publisher can disseminate an event *e* using this primitive;

***notify*(*e*)** A subscriber can receive an event *e* which represents a topic update from the publish-subscribe system;

***subscribe*(*t*)** A client can subscribe to a specific topic *t* using this primitive;

***unsubscribe*(*t*)** A client can unsubscribe from a specific topic *t* using this primitive;

***advertise*(*t*)** A publisher can declare the nature of its future events associated with a topic *t*;

***unadvertise*(*t*)** A publisher can revoke advertisements related to a specific topic *t*.

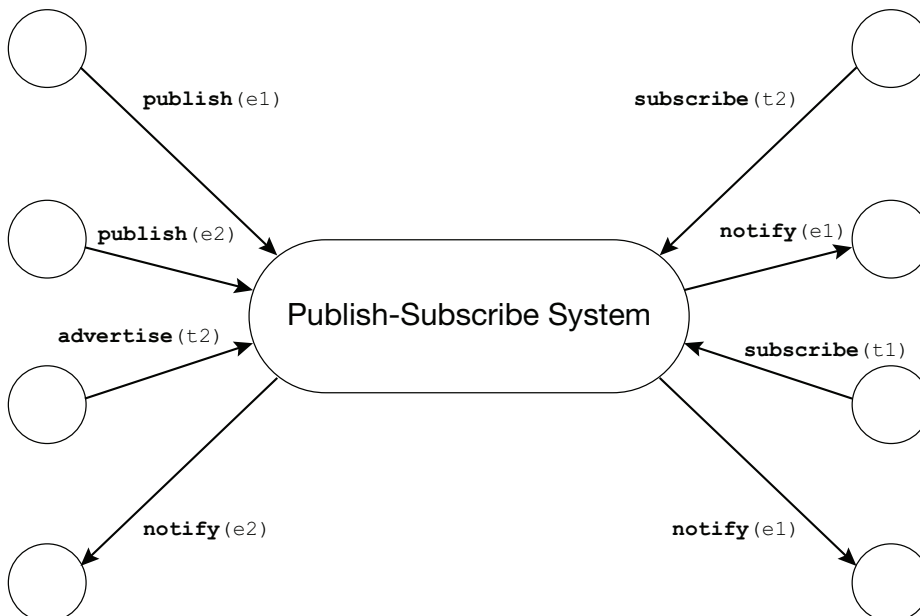


Figure 3.2: The publish-subscribe paradigm

These operations are mediated by a **publish-subscribe system** (a.k.a. **broker**) which is responsible to dispatch the events from publishers to interested subscribers.

Centralized and **distributed** architecture implementations are available. The simplest approach is the centralized one: in this case, every communication (from publishers to broker or from broker to subscribers) takes place through a series of point to point messages. During its working phase, the broker could be a **bottleneck** if it is present in a single instance. To prevent these situations, we can replace the broker with a *network of brokers* that cooperate to offer the desired functionality and service level.

For further information, see [4].

3.2 Message Queue Telemetry Transport

Message Queue Telemetry Transport (MQTT) is a lightweight broker-based publish-subscribe messaging protocol, developed and designed for constrained devices and low bandwidth by *IBM / Eurotech* in 1999. Due to simplicity and low overhead, this protocol is suitable for use in constrained environments, such as:

- Expensive networks with low bandwidth and no reliability;
- Embedded in devices with limited processor and/or memory resources.

Some other features of this protocol are:

- It provides *one-to-many* message distribution and decoupling of applications;
- It is *agnostic* on the content of the payload;
- It is built over TCP/IP for basic network connectivity;
- It provides three different type of *Quality of Service*:

At most once: messages are delivered according to the best effort of TCP/IP networks; Message loss and duplication can occur.

At least once: messages are assured to arrive but duplicates may occur.

Exactly once: messages are assured to arrive exactly once.

- It has a small transport overhead;
- It has a mechanism to notify of abnormal disconnection of a client using the *Last Will* and the *Testament* features.

The protocol is openly published with a royalty-free license and a variety of client libraries have been developed, in particular on popular hardware platforms such as **Arduino**⁴.

⁴<http://arduino.cc>

3.2.1 Message format

The message consists of three parts:

- a fixed header
- a variable header
- a payload.

The message format is depicted in Figure 3.3.

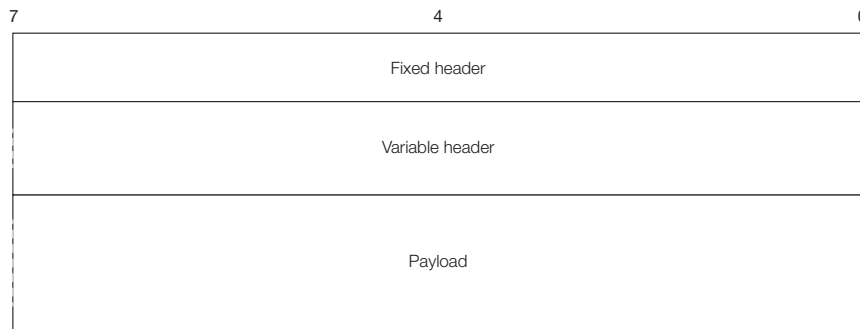


Figure 3.3: Message format.

Fixed header

The fixed header contains information about the **message type**, **QoS level**, some flags and the message length. Figure 3.4 shows the fixed header's structure.

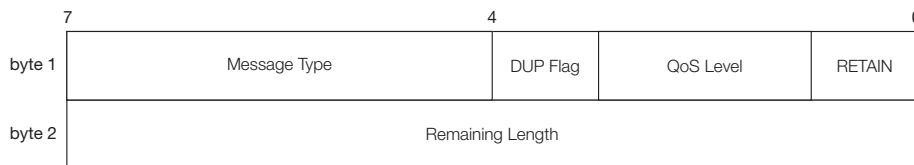


Figure 3.4: Fixed header's bits structure.

Message type This field is used to define the message type. It is a 4 bits field which possible value is listed in Table 3.1.

Mnemonic	Enumeration	Description
<i>Reserved</i>	0	Reserved for future use
CONNECT	1	Client request to connect to broker
CONNACK	2	Connect acknowledgement
PUBLISH	3	Publish message
PUBACK	4	Publish message acknowledgement
PUBREC	5	Publish received (QoS = 2)
PUBREL	6	Publish release (QoS = 2)
PUBCOMP	7	Publish complete (QoS = 2)
SUBSCRIBE	8	Client subscribe request
SUBACK	9	Subscribe acknowledgement
UNSUBSCRIBE	10	Client unsubscribe request
UNSUBACK	11	Unsubscribe acknowledgement
PINGREQ	12	Ping request
PINGRESP	13	Ping response
DISCONNECT	14	Client disconnection request
<i>Reserved</i>	15	Reserved for future use

Table 3.1: Possible message types.

DUP Flag This flag is set when client or server attempts to re-deliver a PUBLISH, PUBREL, SUBSCRIBE or UNSUBSCRIBE message. The flag is used only on messages that have QoS > 0.

QoS level This flag is used to set the level of delivery assurance for PUBLISH messages. The possible QoSs are listed in Table 3.2.

QoS value	Bit 2	Bit 1	Description
1	0	0	At most once (i.e. fire and forget)
2	0	1	At least once (i.e. acknowledged delivery)
3	1	0	Exactly once (i.e. assured delivery)
<i>Reserved</i>	1	1	Reserved for future use

Table 3.2: Possible QoS levels.

Retain This flag is used only on PUBLISH messages. If this flag is set, the broker should hold the message after it has been delivered to the current subscribers. When a new subscription is established on a topic, the last retained messages should be sent to the subscriber with the RETAIN flag set. If there is not retained messages, nothing happen.

Remaining length This byte represent the bytes remaining within the current message, including the *variable header* and the payload.

Variable header

Some MQTT command messages contain a variable header which resides between the fixed header and the payload, as shown in Figure 3.3. This header does not count inside the *Remaining length* field value.

For further information about the prococol, see [11].

3.2.2 Message flows

As we have seen in the previous section, MQTT permits different levels of message delivery according to corresponding *Quality of Service*.

The lower *QoS* is the so called **at most once delivery**: the message is delivered with the **best effort** delivering level of the underlying TCP/IP network. In the protocol, there is not re-transmission concept. The message can either arrive to the broker once or not at all. A time chart is described in Figure 3.5.

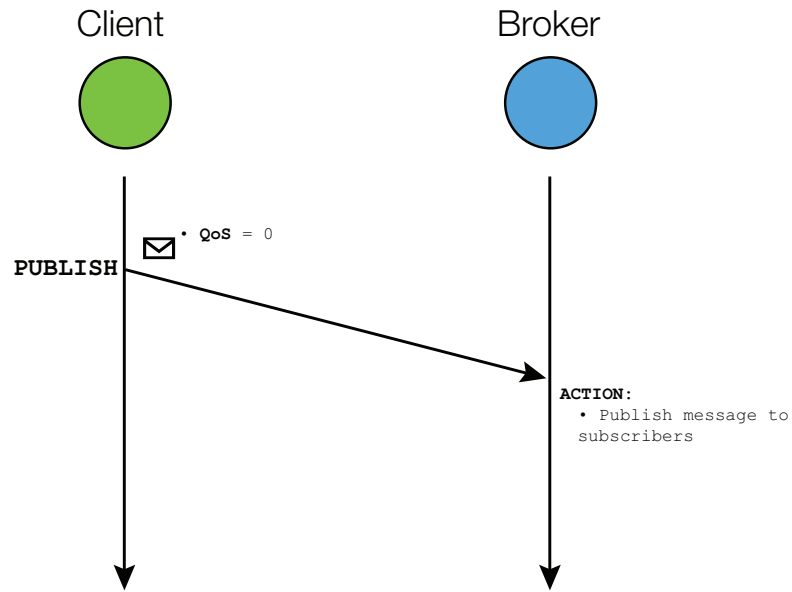


Figure 3.5: Communication flow with QoS level 0

The second *QoS* level is called **at least once delivery**: with respect to the previous *QoS*, every message is uniquely identified by an ID and every correctly received message is confirmed by an acknowledge, using the **PUBACK** primitive. If during the communication a failure is identified or the ACK is not received after a specified period of time, the client sends again the message with the DUP flag set in the fixed header. **SUBSCRIBE** and **UNSUBSCRIBE** messages are sent with *QoS* level 1. Figure 3.6 shows a time diagram.

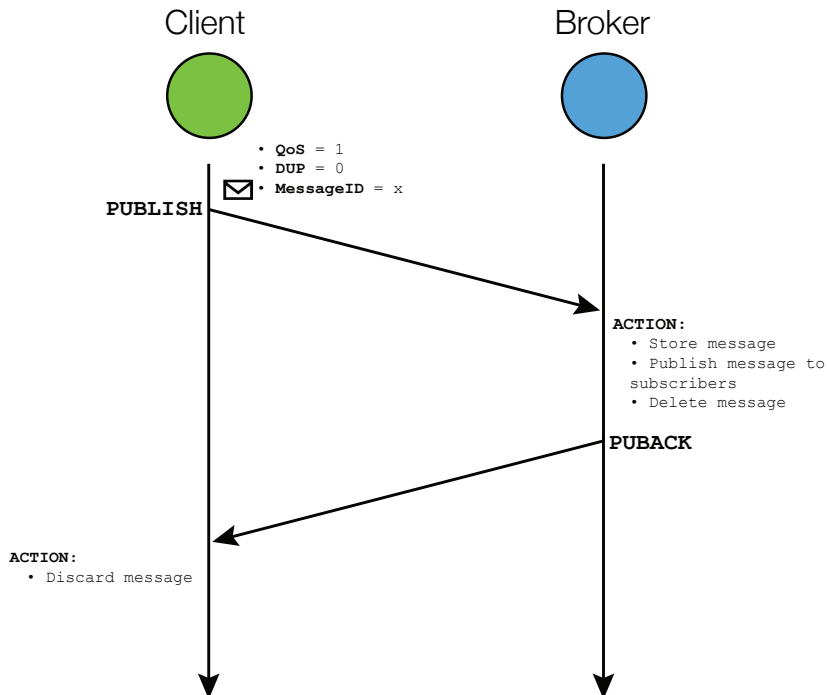


Figure 3.6: Communication flow with QoS level 1

The third *QoS* level guarantees that duplicate messages are not delivered to the receiving application. This level increases the network traffic but it could be acceptable when applications require the correct receipt of the message due to the importance of the message content. As in the previous level, messages are uniquely identified by an ID. Figure 3.7 shows a time diagram.

3.2.3 MQTT Applications

In this section, we present the most significant solution examples of MQTT protocol.

The first one regards medical assistance and it involves the St. Jude Medical with its **home pace-maker monitoring**. The solution consists on home monitoring appliance that publishes diagnostics to health care provider through patient home connection. The main characteristics are:

- **Enabled** higher level of patient care, early diagnostic of problems, peace of mind;
- **Improved** administrative efficiency and maintenance;
- **Helped** conform to standards and eased integration of data.

Another solution example comes from **Consert**, an energy management company. It built an intelligent utility network offering for smart energy. The main characteristics are:

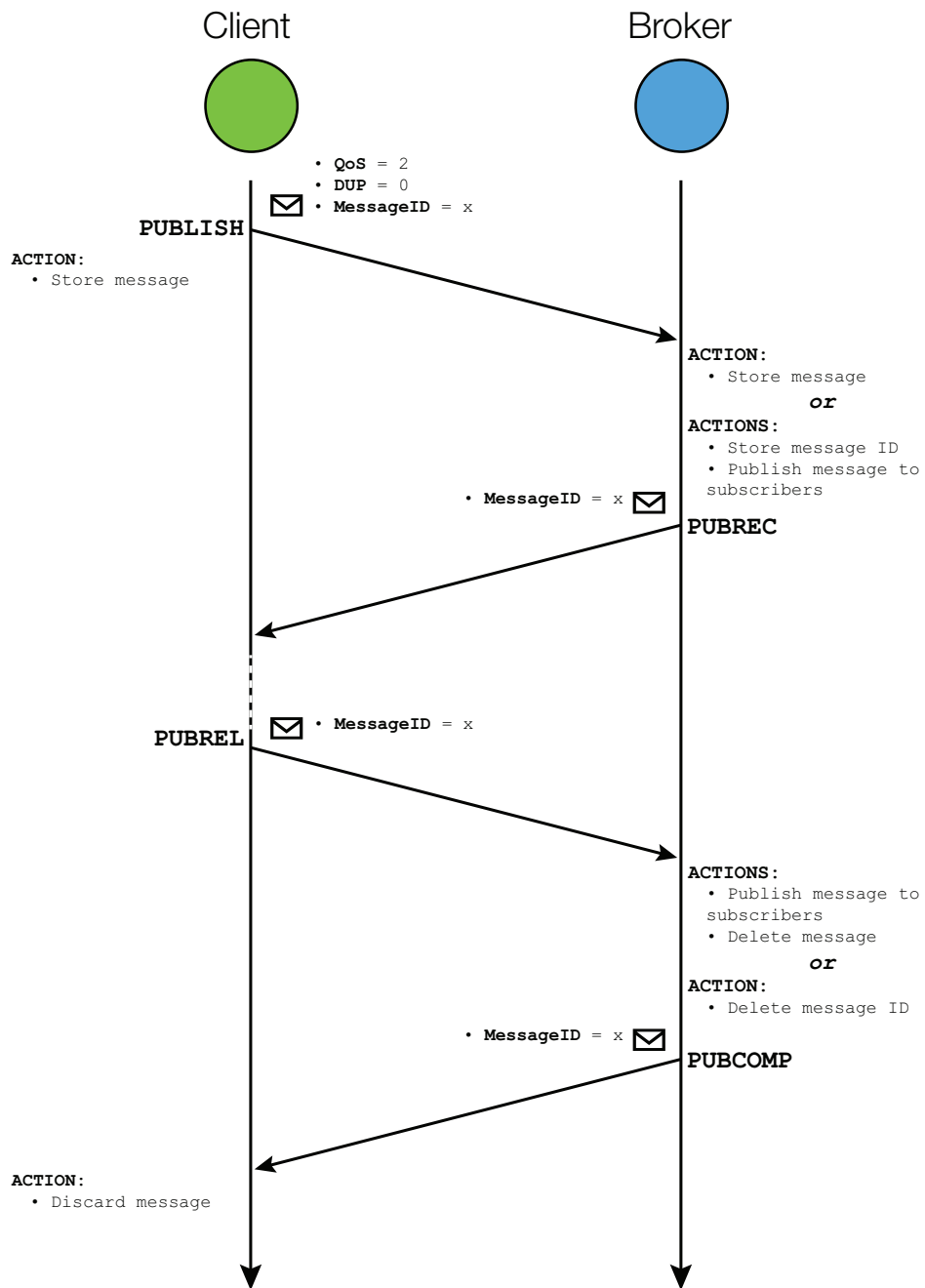


Figure 3.7: Communication flow with QoS level 2

- **Enabled** daily energy savings of 15-20%;
- **Improved** peak usage and avoided over charges;
- **Helped** optimize energy use.

In the social networks era, Facebook [8] used MQTT in its personal messaging app for mobile subscribers [14]. Facebook experiences some problem of *long latency* when sending a message. The method they were using was reliable but it was really slow. They were trying to build up a mechanism that maintains a persistent connection to their servers. In order to do so without killing battery life, Facebook development team used MQTT that they experimented in Beluga. By maintaining an MQTT connection and routing messages through their chat pipeline, Facebook was able to often achieve phone-to-phone delivery in *hundreds of milliseconds*, rather than multiple seconds. The main characteristics are:

- **Enabled** reliable communications between individuals;
- **Improved** delivery times over high latency connections;
- **Helped** improve mobile battery life.

3.3 An MQTT implementation: Paho

Paho is a project which has been created to provide scalable open-source implementations of open and standard messaging protocols aimed at new and existing M2M and IoT applications. Paho is a *Maori* word which means to broadcast, make widely known, announce, disseminate, transmit⁵. Figure 3.8 shows the Paho project logo.



Figure 3.8: Paho project logo.

The project started with MQTT publish/subscribe client implementation but, in the future, it will bring corresponding broker as established by the developers community.

For further information and future developments about this project, see [10].

3.4 Terracotta[®] Universal Messaging: setting up a local environment

In this section we are going to explain a brief tutorial about how to set up a fully functional system with a broker, publishers and subscribers.

There are several MQTT broker implementations available on the Internet, such as Mosquitto⁶, but in this tutorial we will use **Terracotta Universal Messaging**⁷ (Terracotta UM), developed by *Software AG* (SAG). Terracotta UM is

⁵<http://www.maoridictionary.co.nz/search?keywords=paho>

⁶<http://mosquitto.org/>

⁷<http://terracotta.org/products/universal-messaging>

an all-in-one solution for messaging standards, compatible with a considerable number of messaging protocols (e.g. MQTT and JMS). SAG also provides different languages APIs such as Java, C++, .Net, Javascript and mobile (iOS and Android), as shown in Figure 3.9.

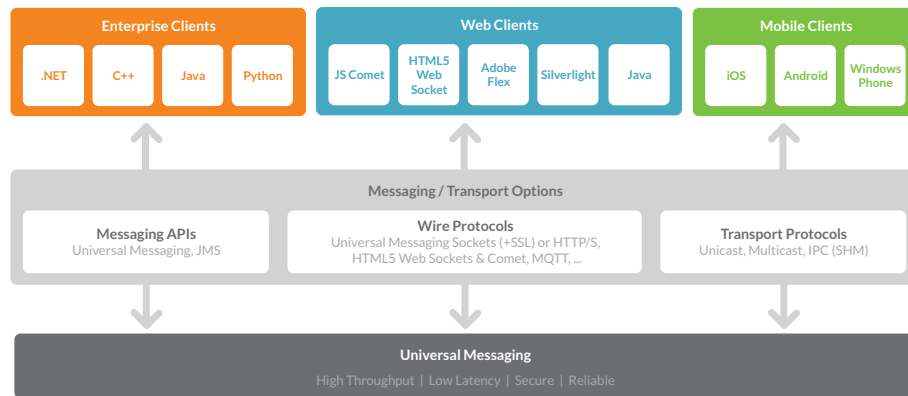


Figure 3.9: Terracotta Universal Messaging Architecture.

3.4.1 Terracotta Universal Messaging

The system deployment has been done on a Linux machine; Terracotta UM requires a Java JDK installation. The first thing to do is download the broker available on the SAG site and install it on your own PC. Once downloaded the Linux version, it requires the permission of execution which can be set with the following commands

```
root# cd <path to downloaded terracotta um bin>
root# chmod 777 ./universalmessaging_linux.bin
```

Now, the downloaded file has permission to execute, so you can start the installation process through the command

```
root# sudo ./universalmessaging_linux.bin -i gui
```

Using the switch `-i gui`, you can see the process via a the OS graphical user interface and not in the terminal console. Following the installation process, you can choose location where Terracotta UM will be; in our case, the selected location is `/bin/terracotta`.

Once Terracotta UM is installed, you can start the messaging server: to do so, type this line in the terminal

```
root# cd /bin/terracotta/universalmessaging_<v_number>
      /links/Server/nirvana
root# sudo ./Start Universal Messaging Realm Server
```

Typed these command, server boots and it will listen on the address `http://0.0.0.0:9000`. The default protocol does not permit MQTT communication, so we have to add an interface which guarantees compatibility with MQTT. In order to do this, we have to start the **Terracotta UM Enterprise Manager**, using the commands:

```

root# cd /bin/terracotta/universalmessaging_<v_number>
      /links/Administration/nirvana
root# sudo ./Universal Messaging Enterprise Manager

```

The Manager interface is shown in Figure 3.10.

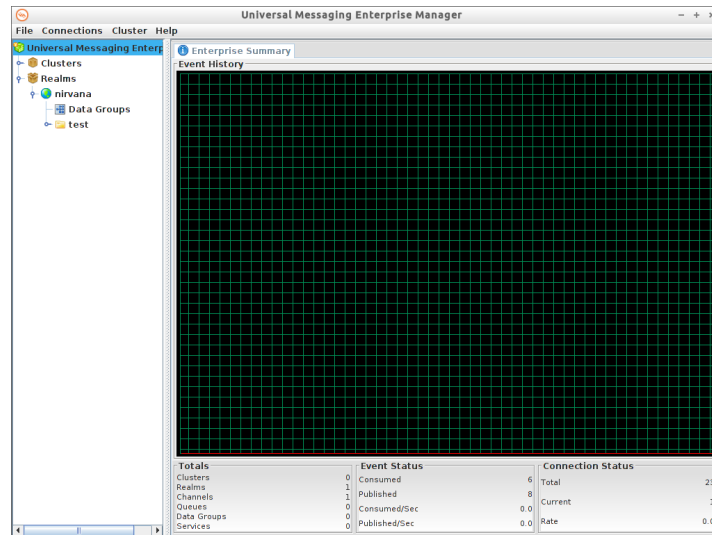


Figure 3.10: Universal Messaging Enterprise Manager.

It provides a graphical representation of statistics about memory usage, messaging and connections status. If you want to manage the MQTT broker, you have to go to the **nirvana** section on the manager's left side. The section is shown in Figure 3.11.

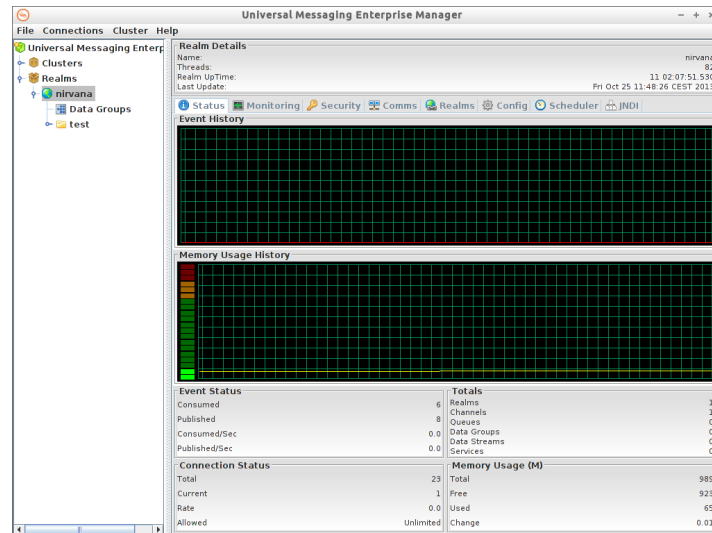


Figure 3.11: Nirvana Manager Section

As we said before, the default interface is listening on **Universal Messaging HTTP Protocol (nhp)** which is not compatible with MQTT. In order to make it compatible, we will add new interface on the nirvana realm. Go to *Comms* tab and then to *Interfaces* sub-tab. The presented interface is shown in Figure 3.12

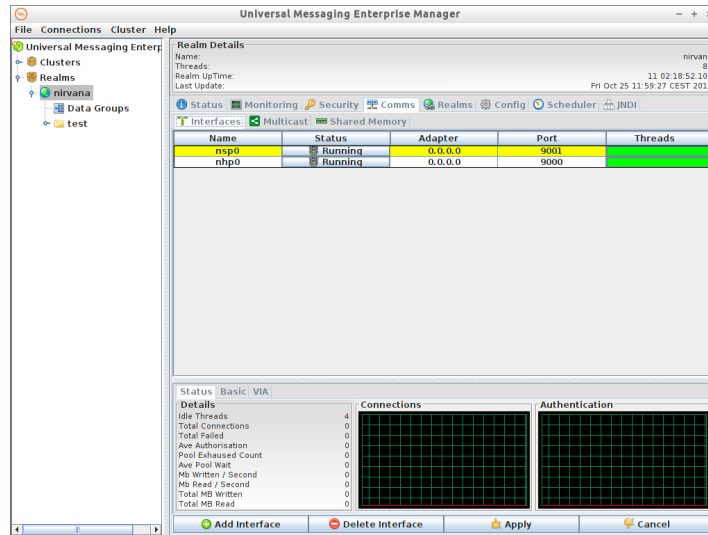


Figure 3.12: New Interface Section.

The UM MQTT-compatible protocol is **Universal Messaging Socket Protocol (nsp)**. Add a new interface, clicking on *Add interface* button and fill the form in Figure 3.13 with the following parameters:

Interface Protocol NSP (Socket Protocol)

Interface Port 9001 (any port > 1024)

Interface Adapter 0.0.0.0 (localhost)

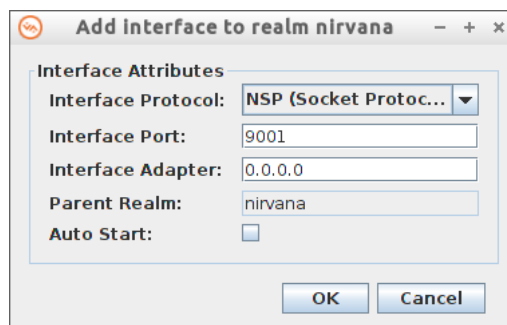


Figure 3.13: New interface parameters.

Once pressed the confirm button, start the interface by clicking on the related button on the Status column.

In order to add custom topics to which clients can subscribe to or publish in, right-click the nirvana icon on the left sidebar and then to *Add Channel*. The presented form is shown in Figure 3.14.

Figure 3.14: New channel form.

We fill it with the desired *Channel Name* (`test/dev`) and leaving blank the others fields.

The created channel is added on the left sidebar, as shown in Figure 3.15, but it does not have any read/write permissions.



Figure 3.15: New channel icon on the left sidebar.

In order to add permissions to the channel, select the `dev` channel, go to *ACL* tab and click the *Add* button. The presented interface requires to add the **user** and the **host** of the related permission with a specified syntax (`user@host`), as shown in Figure 3.16. In our example, we do not set any particular user, so we add the ACL which permits to every users on localhost to access this channel; the specific string is `*@localhost`.

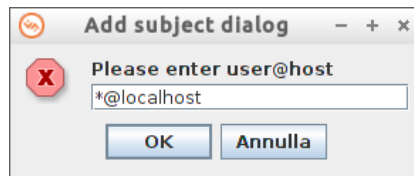


Figure 3.16: New channel permission form.

Once confirmed, a row appears in the *ACL* tab as shown in Figure 3.17. Click *Full* section to allows users subscription, publishing, naming, ACL management and purging.

Subject	Manage ACL	Full	Purge	Subscribe	Publish	Named
*@localhost	✓	✓	✓	✓	✓	✓

Figure 3.17: List of channel's permission.

These are the basic steps to do in order to set up the Terracotta UM server. In the next section we will present a simple implementation of two clients, one will work as *publisher* and another one will work as *subscriber*.

3.4.2 Client Implementation with Paho MQTT

In this section we will provide a simple implementation of a publisher and a subscriber which can exchange messages each other. During this example we will keep separate the publisher and the subscriber functionalities to better explain how they work. Nevertheless, a client can be both publisher and subscriber at the same time.

Publisher

Since the publisher is a MQTT entity, it has to have a `MqttClient` in order to exchange information with the broker. To do so, we instantiate a new `MqttClient` in line 22 of Appendix A

```
MqttClient publisher = new MqttClient(TCPAddress, clientId);
```

where `TCPAddress` and `clientId` are the broker's address and the unique client identifier across the system, respectively.

We connect the client to the broker using the instance method

```
publisher.connect();
```

which can accept an object containing connection configuration options but, in this example, we will use default configuration values.

Now, we are connected to the broker and we can publish on topics. To do so, we retrieve from the `publisher` object a `MqttTopic`

```
MqttTopic t = publisher.getTopic(topic);
```

through which we can send messages to the specific `topic`.

Sending information are wrapped by a class that provides several methods to work with. The message quality of service level can be set using the `MqttMessage` method

```
message.setQoS(QoS);
```

Messages can be sent using the topic previously get and invoking on it the `publish` method

```
MqttDeliveryToken token = t.publish(message);
```

which returns a `MqttDeliveryToken` containing communication information. Both synchronous and asynchronous waiting can be possible: the synchronous waiting can be done using the `MqttDeliveryToken`, returned by the `publish` method, and invoking on it the method

```
token.waitForCompletion(timeout);
```

which accept a timeout parameter that represents the maximum amount of time MQTT client will wait for publishing completion.

The complete publisher implementation can be found in Appendix A.

Subscriber

The MQTT subscriber is quite simple to the publisher with the exception that subscriber needs callbacks for managing MQTT events, i.e. arrival of a new message, connection loosing and delivery completion. There are two ways to attach these callbacks to client:

- create a new class that implements the `MqttCallback` interface;
- implements `MqttCallback` interface on the same subscriber class.

In this example we chose to create a new class which implements the interface. `SubscriberCallbacks` contains three methods that have to be override in order to make the class compliant to the interface. These three methods are:

- `messageArrived`: it shows the arrived message on the standard output;
- `deliveryComplete`: it shows the token information associated with the delivery;
- `connectionLost`: it prints several information about the exception which causes the connection lost.

Once implemented the class, we can attach it to the subscriber through the code

```
SubscriberCallbacks callbacks =
    new SubscriberCallbacks(clientId);
subscriber.setCallback(callbacks);
```

The complete publisher implementation can be found in Appendix B.

Chapter 4

Integrating MQTT into the platform

Since the platform described in Chapter 2 implements a *Request/Response* (R/R) architecture, integrating MQTT into it means inserting a new player into the system: the **broker**.

With respect to the R/R paradigm, where ON communicates directly with SP, in a publish/subscribe (P/S) architecture, each communication is mediated by the broker, as described in Figure 4.1.

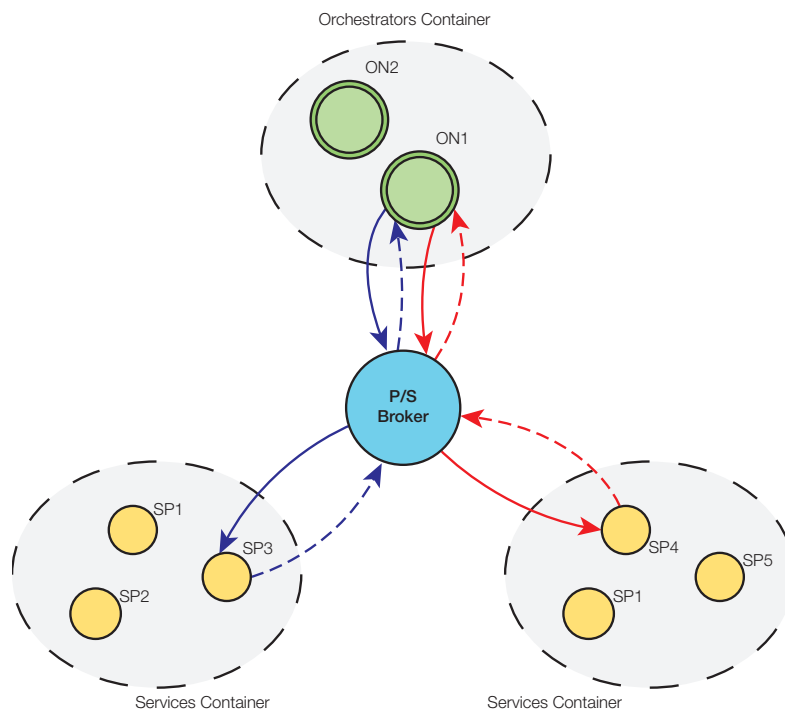


Figure 4.1: System architecture with P/S broker.

It has decided to include a P/S system, and in particular MQTT, in the project in order to integrate into the platform, not only web services but also **physical devices** which can exchange information from (e.g. sensors) and to (e.g. actuator) the real world.

In the following sections, we will describe two different integration of P/S paradigm to the platform illustrated in [15]: a **centralized** and a **distributed** implementation.

4.1 Centralized SEP

In the centralized implementation, all the concepts presented in [15] has been kept: the ON is the only that knows mashups logic, SPs execute in the way described in 2.2.

The information carried by the session message, from ON to SP, are almost the same as the ones described in 2.2: **ActionId**, **TargetSPId**, **SessionId**, **MashUpId** and **InParameters**.

- **ActionId**: the action identifier we want to invoke on receiver SP;
- **TargetSPId**: this parameter is used to manage the case in which a mashup contains multiple instances of a single SP (e.g. `<SP_ID>.0` or `<SP_ID>.1`);
- **SessionId**: the identifier of a particular execution of a mashup;
- **MashUpId**: the mashup identifier;
- **InParameters**: the container that holds the input parameters related to a specific SP;
- **OutParameters**: the container that holds all the output parameters produced by SPs.

Communications between SPs and ON take place through the P/S concept of **topic**. Each SP can receive messages subscribing itself to a specific topic which has the form `/<SP_ID>/execute`, where `<SP_ID>` is the SP identifier (e.g. `/GMail/execute` can be the topic for a mail SP). Once received the message, each SP has to dispatch the invoking action using the information carried by the session message.

On the other hand, ON can receive information from SP subscribing to `/ON/notify`: each SP, at the end of action execution, will publish on it in order to notify ON.

4.1.1 Example

In order to better explain how the platform works, we get to the reader a practical example and we will describe every step in the mashup execution. Figure 4.2 represents a simple composition of SPs.



Figure 4.2: An example of mashup.

SPs can exchange output parameters between them, i.e. output parameters of a SP can be input parameters of another one: in SCP, the mashup creator can define these connections placing a **placeholder** as parameter value (e.g. `SPMail.0.in.body = ${LightSensor.0.out.intensity}` will set the mail's body to intensity value outputted by the light sensor).

Deployment

The deployment phase of a mashup consists of retrieving its XML representation stored in the repository, generate the routing table and keep it in memory. The table generated for mashup in Figure 4.2 is described in Table 4.1:

Publisher	Subscriber	Topic	Parameters
START	SP1.0	SP1.0/start	
SP1.0	SP2.0	SP1.0/t1	"SP1.0.in.prop1" : 0.5, "SP1.0.in.prop2" : "sword"
SP2.0	SP3.0	SP2.0/t3	"SP2.0.in.prop1" : "\${SP1.0.out.prop1}", "SP2.0.in.prop2" : 4
SP3.0	END	SP3.0/t5	"SP3.0.in.prop1" : "\${SP1.0.out.prop3}", "SP3.0.in.prop2" : "\${SP2.0.out.prop2}"

Table 4.1: Routing table for mashup in Figure 4.2.

Execution

Execution steps are depicted in Figure 4.3.

Since the ON manages the entire mashup execution, when the ON receives a SP's session message, it looks up the routing table, it retrieves the SP (possibly SPs) that has to be execute afterwards, modifying the session message.

1. START SP creates the session message in Listing 1 and publishes it to `/ON/notify`:

Listing 1 Initial session message JSON representation.

```
1 {
2   "SessionId" : <Random_ID>
3   "MashUpId" : "MS1"
4   "ActionId" : "start",
5   "TargetSPId" : "START",
6 }
```

2. The only SP that has to be started is SP1.0, so, ON modifies the session message, which is described in Listing 2:

Listing 2 Session message JSON representation sent to SP1.

```
1 {
2   "SessionId" : <Random_ID>
3   "MashUpId" : "MS1"
4   "ActionId" : "t1",
5   "TargetSPId" : "SP1.0",
6   "InParameters" : {
7     "SP1.0.in.prop1" : 0.5,
8     "SP1.0.in.prop2" : "sword"
9   }
10 }
```

This message is published to /SP1/execute.

3. When SP1 ends, the result of its execution is appended to the current session message, published to /ON/notify and received by the ON. The published session message is described in Listing 3:

Listing 3 Session message JSON representation modified by SP1.

```
1 {
2   "SessionId" : <Random_ID>
3   "MashUpId" : "MS1"
4   "ActionId" : "t1",
5   "TargetSPId" : "SP1.0",
6   "InParameters" : {
7     "SP1.0.in.prop1" : 0.5,
8     "SP1.0.in.prop2" : "sword"
9   }
10  "OutParameters" : {
11    "SP1.0.out.prop1" : 1.25,
12    "SP1.0.out.prop3" : "joke"
13  }
14 }
```

4. Once the ON receives the session message in Listing 3, it looks up in the routing table for finding what to do. From Table 4.1, using `TargetSPId` as key, ON retrieves that `SP2.0` is the next SP to invoke. ON checks the parameters container and finds that it contains a placeholder (i.e. `#{SP1.0.out.prop1}`): with this placeholder, it goes to `OutParameters` field of the current session message and retrieves the value (e.g. 1.25). Afterwards, the session message published to `/SP2/execute` is listed in Listing 4:

Listing 4 Session message JSON representation published to SP2.

```

1  {
2      "SessionId" : <Random_ID>
3      "MashUpId" : "MS1"
4      "ActionId" : "t3",
5      "TargetSPId" : "SP2.0",
6      "InParameters" : {
7          "SP2.0.in.prop1" : 1.25,
8          "SP2.0.in.prop2" : 4
9      }
10     "OutParameters" : {
11         "SP1.0.out.prop1" : 1.25,
12         "SP1.0.out.prop3" : "joke"
13     }
14 }

```

5. Once again, SP2 appends to the `OutParameters` its execution results and published session message is listed in Listing 5:

Listing 5 Session message JSON representation published to ON by SP2.

```

1  {
2      "SessionId" : <Random_ID>
3      "MashUpId" : "MS1"
4      "ActionId" : "t3",
5      "TargetSPId" : "SP2.0",
6      "InParameters" : {
7          "SP2.0.in.prop1" : 1.25,
8          "SP2.0.in.prop2" : 4
9      }
10     "OutParameters" : {
11         "SP1.0.out.prop1" : 1.25,
12         "SP1.0.out.prop3" : "joke",
13         "SP2.0.out.prop2" : 3.1415
14     }
15 }

```

6. ON receives this session message and applies the same rule used before, it

produces the new session message, which will be published to `/SP3/execute`, in Listing 6:

Listing 6 Session message JSON representation published to SP3.

```

1  {
2      "SessionId" : <Random_ID>
3      "MashUpId" : "MS1"
4      "ActionId" : "t5",
5      "TargetSPId" : "SP3.0",
6      "InParameters" : {
7          "SP3.0.in.prop1" : "joke",
8          "SP3.0.in.prop2" : 3.1415
9      }
10     "OutParameters" : {
11         "SP1.0.out.prop1" : 1.25,
12         "SP1.0.out.prop3" : "joke",
13         "SP2.0.out.prop2" : 3.1415
14     }
15 }

```

7. SP3 executes and appends its results to the `OutParameters`. The resulting session message (which will be published on `/ON/notify`) is listed in Listing 7:

Listing 7 Session message JSON representation published to ON by SP3.

```

1  {
2      "SessionId" : <Random_ID>
3      "MashUpId" : "MS1"
4      "ActionId" : "t5",
5      "TargetSPId" : "SP3.0",
6      "InParameters" : {
7          "SP3.0.in.prop1" : "joke",
8          "SP3.0.in.prop2" : 3.1415
9      }
10     "OutParameters" : {
11         "SP1.0.out.prop1" : 1.25,
12         "SP1.0.out.prop3" : "joke",
13         "SP2.0.out.prop2" : 3.1415,
14         "SP3.0.out.prop2" : "goo.gl/Bf51X4"
15     }
16 }

```

8. ON receives the session message sent by SP3 and it checks what is going to do. SP3 is the last block in the composite service and therefore ON publishes to `END/execute` the session message just received. This event ends the mashup execution.

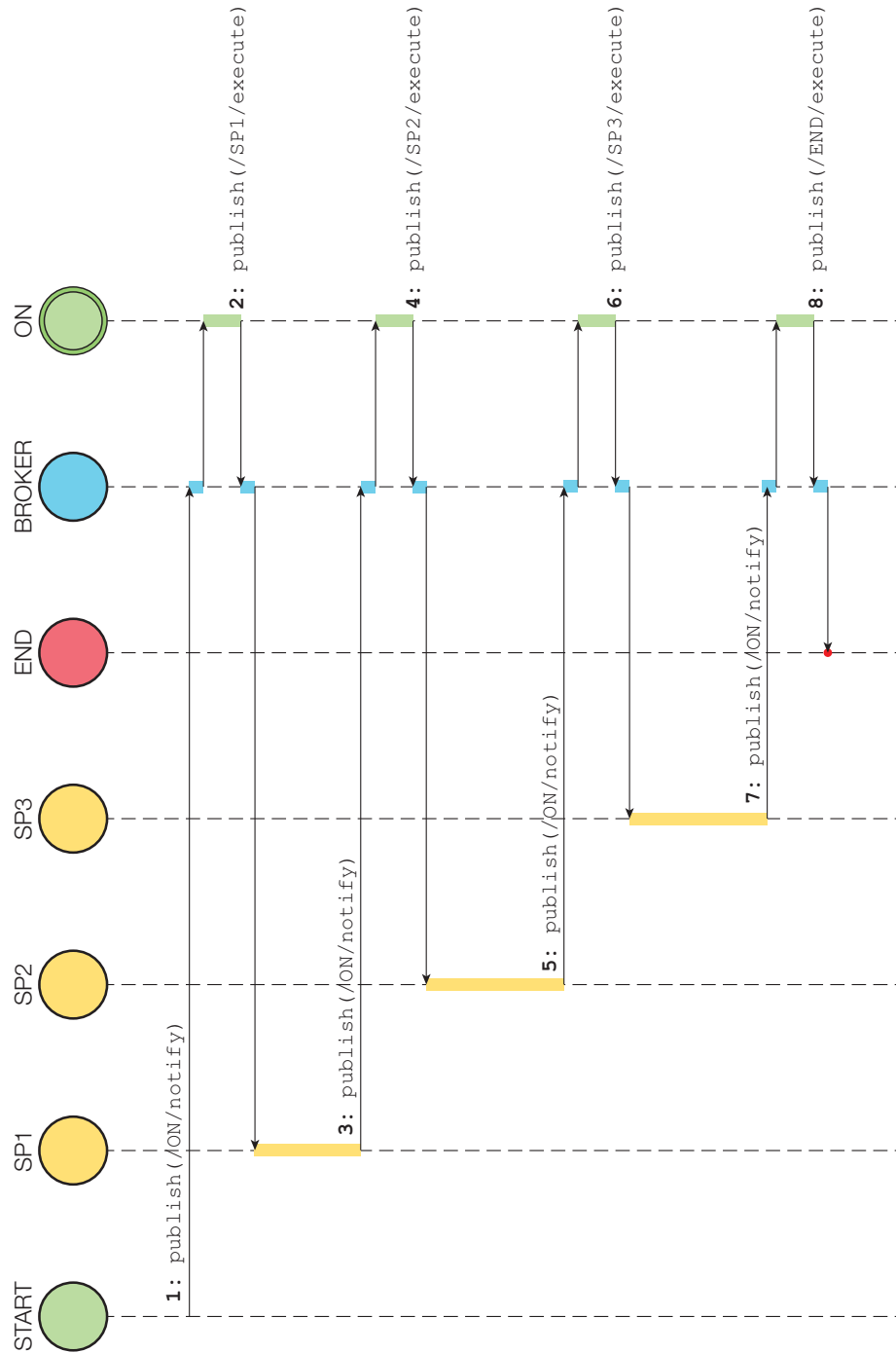


Figure 4.3: Centralized SEP communication diagram.

4.2 Distributed SEP

The distributed SEP differs from the centralized one for several reasons. The mashup execution is not managed by a central node but every SP is responsible for a part of the SPs chain.

The orchestrator node is replaced by a new entities that we called **deployment manager** (DM). Starting from the same table structure in Table 4.1, DM creates special messages, called **deployment messages** which are sent to the SP during the deployment phase. Since the mashup logic is distributed on the SPs, there is the need to identify multiple deployed mashups and to manage multiple instances of the same SP in a mashup. To do so, we have introduced a **token** which identifies uniquely each edge in the mashup graph representation.

Different topics are associated to each SP: each SP has as many topics as the actions available, moreover, it has some **administration topics** through which SP can be notified of deployment or undeployment events.

The next two sections explain in a detailed way how deployment and execution phases work.

Deployment

As said in the introduction of distributed SEP, we introduced a new message called **deployment message**. The deployment message is based on the same routing table structure described in Table 4.1.

A deployment message contains:

- **Mashup Id:** this field identifies mashup which the deployment message belongs to;
- **SP Counter:** this field is used to distinguish multiple repetitions of the same SP in a mashup;
- **Subscription Topic:** this is the topic to which SP has to subscribe in order create the chain;
- **Publishing Topic:** this is the topic on which SP has to publish in order to create the chain. This represents also the *action* that will be executed on the SP;
- **Subscription Token:** this is the random token that SP has to receive from the previous SP;
- **Publishing Token:** this is the random token that SP has to send to the next SP;
- **In Parameters:** this is a collection of SP input parameters used by the service. This collection can contain parameters whose values are **placeholders**.

In this scenario, a SP can receive multiple deployment messages, one message for each SP instance in the mashup.

Once DM published these messages on the SP's administration topics, each SP maintains a special data structure that facilitate deployed mashup administration. A SP uses two **maps** in order to maintain and manage the users' requests during the mashup execution:

- **Mashups Map:** this map contains one element for each deployed mashup on the platform. A map element is composed by a *key* and a *value*; it has the *Mashup Id* as a key and a **list** of subscription tokens as value. Since a single SP can be added multiple times in a mashup, therefore it can be located in different places inside the composite service, the list represents all instances of the same SP that are in the mashup;
- **Tokens Map:** this map contains one element for every single SP instance in the mashup. It has the *subscription token* as key and a object representing the action configuration information required for the SP execution.

When a new deployment message arrives, SP updates its maps, i.e. it adds the *mashup id* to the *mashups map*, if it does not exists, and adds the *subscription token* to the correspondent list. Once updated the *mashups map*, SP adds the action's information object to *tokens map* using *subscription token* as key; the information object will contain *publishing topic*, *publishing token*, *subscription topic*, *SP counter*, *in parameters* and *action type*. Then, it subscribes itself to the *subscription topic* defined in the just received deployment message.

In addition to general SPs, the platform needs two special SP: a **START SP** which assigns a **session id** and starts a mashup execution, and a **END SP** which collects all the final SP results.

After the deployment phase, all the mashup execution is managed by each SP and there is no need of any external entity operation.

Execution

During the execution of a mashup, SP receives from a topic and publishes on another one. All the information about a specific mashup execution is stored in a **session message** and it travels along the composite service saving all the output values. A session message contains:

- **Mashup Id:** this field identifies which mashup is running;
- **Session Id:** this field identifies uniquely a mashup execution;
- **Token:** this field identifies uniquely a connection between two SP;
- **Out Parameters:** this container holds all the SP's results, from the beginning to the end.

Throughout the execution, every SP will modify the *token* session property with the token that next SP expects and add to the *out parameters* the SP execution result. An example will better explain how this mechanism works.

4.2.1 Example

The SP configuration is the same proposed in Section 4.1.1.

Deployment

The communication diagram is depicted in Figure 4.4.

1. Starting from the routing table in Table 4.1, DM assigns different random tokens to each connection. In case of multiple listeners on the same topic, the publishing token will be the same for specific connections. Table 4.2 summarizes the tokens associated with each connection.

Publisher	Subscriber	Topic	Token
START	SP1.0	SP1.0/start	\mathcal{T}_1
SP1.0	SP2.0	SP1.0/t1	\mathcal{T}_2
SP2.0	SP3.0	SP2.0/t3	\mathcal{T}_3
SP3.0	END	SP3.0/t5	\mathcal{T}_4

Table 4.2: Communication links tokens.

The deployment process will produce the following deployment messages:

Listing 8 START SP deployment message.

```

1 {
2     "MashUpId" : "MS1",
3     "PublishingTopic" : "SP1/start",
4     "PublishingToken" : T1,
5     "StartingSP" : "SP1"
6 }
```

Listing 9 SP1 deployment message.

```

1 {
2     "MashUpId" : "MS1",
3     "SPCounter" : 0,
4     "PublishingTopic" : "SP1/t1",
5     "PublishingToken" : T2,
6     "SubscribingTopic" : "SP1/start",
7     "SubscribingToken" : T1
8 }
```

Listing 10 SP2 deployment message.

```

1 {
2     "MashUpId" : "MS1",
3     "SPCounter" : 0,
4     "PublishingTopic" : "SP2/t3",
5     "PublishingToken" : T3,
6     "SubscribingTopic" : "SP1/t1",
7     "SubscribingToken" : T2
8 }
```

Listing 11 SP3 deployment message.

```

1 {
2     "MashUpId" : "MS1",
3     "SPCounter" : 0,
4     "PublishingTopic" : "SP3/t5",
5     "PublishingToken" : T4,
6     "SubscribingTopic" : "SP2/t3",
7     "SubscribingToken" : T3
8 }
```

Listing 12 END SP deployment message.

```

1 {
2     "MashUpId" : "MS1",
3     "SubscribingTopic" : "SP3/t5",
4     "SubscribingToken" : T4
5 }
```

Messages in Listings 8, 9, 10, 11 and 12 will be published on `/START/depoy`, `/SP1/depoy`, `/SP2/depoy`, `/SP3/depoy` and `/END/depoy`, respectively.

Once received by SPs, each one will update its own data structures in order to deploy the mashup on the platform. We presents how the SP1 deployment process works because the others are the same.

SP1 receives deployment message in Listing 9: since *mashup id* MS1 does not exist in the structure, it adds to the *mashups map* a new element with key "MS1" and value a list with one element, i.e. *subscription topic* \mathcal{T}_1 (if another deployment messege were to get for the same mashup, the related *subscription token* will be added to the existing list). \mathcal{T}_1 will be added to *tokens map* as key of a new element and its value will be setted with a new object that contains all the information listed in Section 4.2.

The START and END SPs are a bit different from the others SP: START SP only publishes on topics and END SP only subscribes to topics. Therefore, the data structures maintained by them are easier than the SPs' ones:

START SP has only a map that maintains an object that contains list of first-to-execute SPs and the *publishing token*, associated to the *mashup id* key. On the other hand, the END SP maintains a map that associates to the *mashup id*, an object that contains a list of *subscription tokens*.

Execution

2. Mashup execution starts from START SP: it knows which SPs have to be first executed. In our example, the SP that has to be started is SP1, therefore, START SP creates the session message listed in Listings 13 and it publishes this message on `/SP1/start`.

Listing 13 Initial session message.

```

1  {
2      "MashUpId" : "MS1",
3      "SessionId" : <sessionId1>,
4      "CommunicationToken" : T1
5  }
```

3. When SP1 receives the session message, using `MashupId` and `CommunicationToken`, it retrieves the action to be invoked and the information for session message forwarding. SP1 executes, it modifies the session message as listed in Listing 14 and it publishes it on `SP1/t1`.

Listing 14 Session message forwarded by SP1.

```

1  {
2      "MashUpId" : "MS1",
3      "SessionId" : <sessionId1>,
4      "CommunicationToken" : T2,
5      "OutParameters" : {
6          "SP1.0.out.prop1" : 1.25,
7          "SP1.0.out.prop3" : "joke"
8      }
9  }
```

4. The same procedure takes place when SP2 receives the session message from SP1. Using the `MashupId` and the `CommunicationToken`, it gets the action and, given that the input parameters contain placeholders, SP2 will substitute the actual parameters values from the session message. Therefore, it modifies the session message as in 15 and it publishes on `/SP2/t3`.

Listing 15 Session message forwarded by SP2.

```
1  {
2    "MashUpId" : "MS1",
3    "SessionId" : <sessionId1>,
4    "CommunicationToken" : T3,
5    "OutParameters" : {
6        "SP1.0.out.prop1" : 1.25,
7        "SP1.0.out.prop3" : "joke",
8        "SP2.0.out.prop2" : 3.1415
9    }
10 }
```

5. SP3 receives the session message and it invokes the action. Since the input parameters contain placeholders, SP3 will fetch the actual values from the session message, specifically from the `OutParameters`. Then it modifies the session message as in Listing 16 and it publishes on `/SP3/t5`.

Listing 16 Session message forwarded by SP3.

```
1  {
2    "MashUpId" : "MS1",
3    "SessionId" : <sessionId1>,
4    "CommunicationToken" : T4,
5    "OutParameters" : {
6        "SP1.0.out.prop1" : 1.25,
7        "SP1.0.out.prop3" : "joke",
8        "SP2.0.out.prop2" : 3.1415,
9        "SP3.0.out.prop2" : "goo.gl/Bf51X4"
10    }
11 }
```

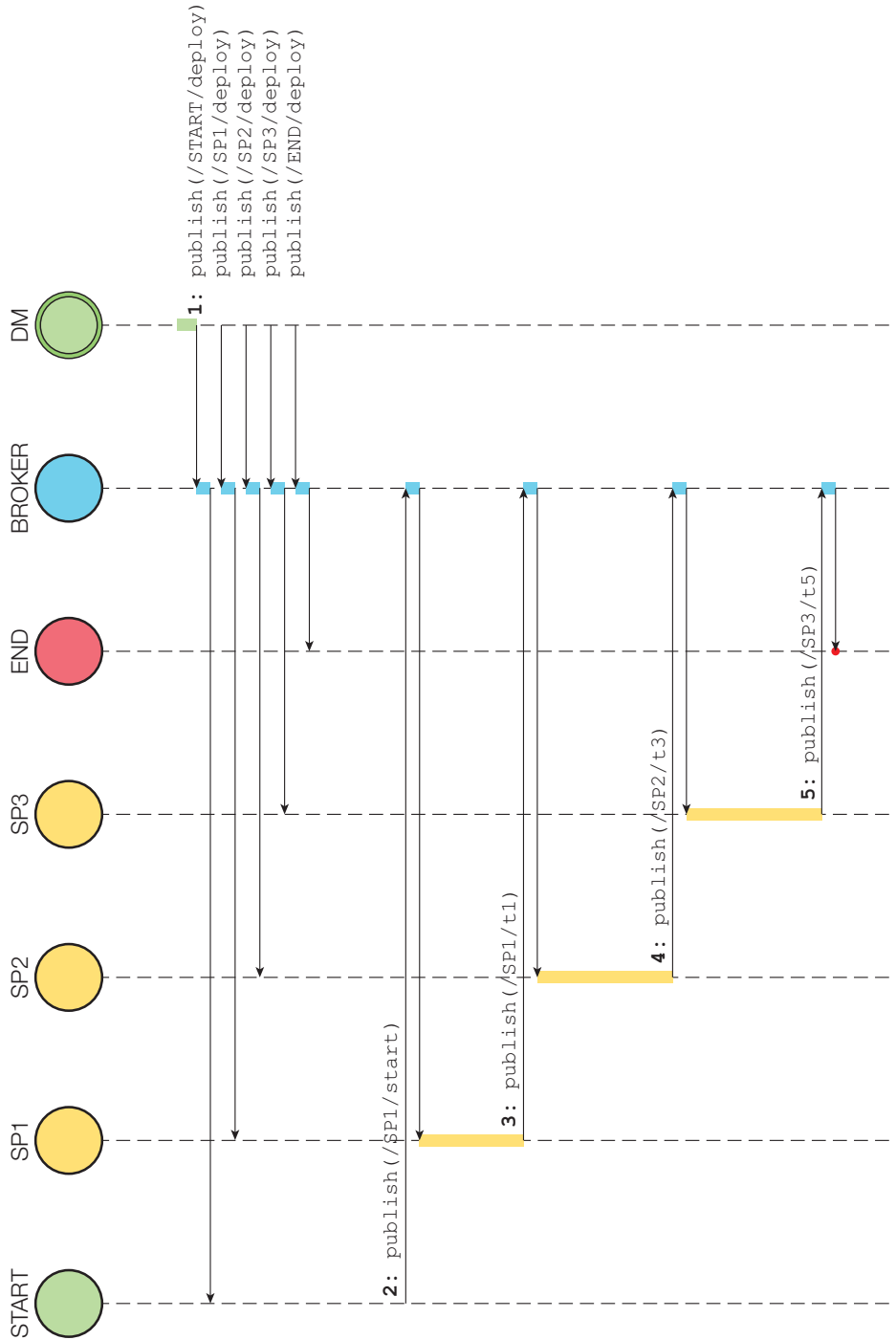


Figure 4.4: Distributed SEP communication diagram.

	Centralized	Distributed
SP	λ requests/s	λ requests/s
ON	$n \times \lambda$ requests/s	n/a
Broker	$2 \times n \times \lambda$ requests/s	$n \times \lambda$ requests/s

Table 4.3: Loading comparison of centralized and distributed implementation.

4.3 Centralized and distributed implementation comparison

Each proposed solution has some pros and some cons. The *distributed* SEP has the main advantage that it scales properly with the number of possible services and devices. If a single SP can not be sufficient due to a large amount of requests, **clusters** of SPs can be created in order to supply a better service level. The run-time bottleneck, that in the centralized version was the *orchestrator*, in this implementation does not exist.

On the other hand, *centralized* SEP makes easy the accounting of resources in a *pay-per-use* manner.

Comparisons can be made on the load each SP could be subject to during its working life cycle. Suppose to have the composite service in Figure 4.2, which has three different SPs (i.e. SP1, SP2 and SP3) and two edges SPs (i.e. START and END).

If the system is loaded by λ requests/s, in the centralized version each SP will manage λ requests/s (in the hypothesis that a single instance exists in the mashup, otherwise $m \times \lambda$, where m is the number of instances of a single SP in the mashup) but the *orchestrator node* will be loaded by $n \times \lambda$ requests/s (where n is the number of SPs in the requested mashup). The broker should manage a requests rate of $2 \times n \times \lambda$ requests/s, due to the *request-response* invocation mechanism.

In the distributed version, the SP loading is the same as before but the broker loading reduces to $n \times \lambda$ requests/s, since every SP talks directly to the following SP (possibly, SPs). Table 4.3 summarizes the results.

Another characteristic that can be compared is the **number of topics** created in the two versions. In the centralized one, one topic for each SP is created (i.e. `<SP_ID>/execute`) in addition to the orchestrator node topic (i.e. `ON/notify`). Therefore, the total number of topics that the broker has to manage is $n + 1$, where n is the number of SP available in the platform.

In the distributed version, a topic for each available action in the SP is created plus the administration topics, so, supposing that each SP has m actions, and that n SPs are available in the platform, the broker has to manage $(m + 3) \times n$ topics.

Conclusion and future developments

In the first part of this work, we introduced the *Internet of Things*, a study about the semantic meaning of it and the **Web of Things**, a specialization of IoT. We have seen that some companies built their core businesses on creating platforms that support the emerging Internet of Things, such as **Xively**.

The platform presented in [15] by Stecca and Maresca has been the starting point of the project developed in this thesis.

Enabling devices to be part of this platform has been the focus objective. Not only web services but also smart object which can be combined to create custom services that perform personalized actions.

As we have seen in Chapter 2, the platform proposed in [15] does not scale as desired and, in order to support both devices and web services into it, we rearranged the platform in a **publish-subscribe** way. Two solutions were been proposed, both of them were P/S enabled.

The first one was a translation from the *request-response* to the *publish-subscribe* paradigm: the concepts introduced in [15] were been maintained and adapted to the new architecture, as described in Section 4.1.

The second solution was much more innovative: we removed the **orchestrator node** and introduced the **deployment manager**. We distributed the mashup knowledge into each service proxy, splitting the composite service logic. The *deployment manager* is responsible to distribute the mashup logic knowledge among service proxies, in order to guarantee the correct functionality of the entire composite service.

The implemented platform will be integrated into the iCore project in the future, when more **performance tests** will be attempted.

As part of an evolving project, the application scenarios are continuously changing but the most important concern **smart home**, **smart city**, **smart meeting** and **smart business**.

Another important future development will be the creation of a **Software Development Kit** (SDK) in order to enable third party developers to create custom service proxies and, therefore, enlarge the service proxies available to the platform.

Appendix A

Publisher Java Code

```
1 package mqtt.paho.example1;
2
3 import org.eclipse.paho.client.mqttv3.MqttClient;
4 import org.eclipse.paho.client.mqttv3.MqttException;
5 import org.eclipse.paho.client.mqttv3.MqttTopic;
6 import org.eclipse.paho.client.mqttv3.MqttMessage;
7 import org.eclipse.paho.client.mqttv3.MqttDeliveryToken;
8 import java.util.Scanner;
9
10 public class MQTTPublisher {
11
12     public static String TCPAddress = "tcp://127.0.0.1:9001";
13     public static String topic = "test/dev";
14     public static String payload = "";
15     public static int QoS = 2;
16     public static int timeout = 10000;
17     public static String clientId = "publisher_1";
18
19     public static void main(String[] args) {
20         try {
21
22             MqttClient publisher = new MqttClient(TCPAddress, clientId);
23
24             Scanner reader = new Scanner(System.in);
25             System.out.println("Write messages...");
26             System.out.print("# ");
27             payload = reader.nextLine();
28             publisher.connect();
29             while (!payload.equalsIgnoreCase("$quit")) {
30                 MqttTopic t = publisher.getTopic(topic);
31                 MqttMessage message = new MqttMessage(payload.getBytes());
32                 message.setQos(QoS);
33
34                 MqttDeliveryToken token = t.publish(message);
```

```
35         System.out.println("    Delivery token \""
36                             + token.hashCode());
37         System.out.flush();
38         token.waitForCompletion(timeout);
39         System.out.println("    RECEIVED: "
40                             + token.isComplete());
41
42
43         System.out.print("# ");
44         payload = reader.nextLine();
45     }
46     publisher.disconnect();
47
48 } catch (MqttException e) {
49     e.printStackTrace();
50 }
51 }
52 }
```


Appendix B

Subscriber Java Code

```
1 import org.eclipse.paho.client.mqttv3.*;
2
3 public class SubscriberCallbacks implements MqttCallback {
4     private String instanceData = "";
5
6     public SubscriberCallbacks(String instance) {
7         this.instanceData = instance;
8     }
9
10    @Override
11    public void messageArrived(String topic, MqttMessage message) {
12        System.out.println("Message arrived: " + message.toString());
13        System.out.println("Topic: " + topic.toString());
14        System.out.println("Instance: " + instanceData);
15    }
16
17    @Override
18    public void connectionLost(Throwable cause) {
19        System.out.println("Connection lost on instance " + instanceData +
20            "with cause " + cause.getMessage());
21        System.out.println("Reason code: " + ((MqttException) cause)
22            .getReasonCode());
23        System.out.println("Cause: " + ((MqttException) cause).getCause());
24        cause.printStackTrace();
25    }
26
27    @Override
28    public void deliveryComplete(IMqttDeliveryToken token) {
29        try {
30            System.out.println("Delivery token \"" + token.hashCode() +
31                "\" received by instance " + instanceData);
32        } catch (Exception e) {
33            e.printStackTrace();
34        }
35    }
36 }
```

```
35     }
36
37 }
38
39 import org.eclipse.paho.client.mqttv3.*;
40 import java.util.Scanner;
41
42 public class MQTTSubscriber {
43     public static String TCPAddress = "tcp://localhost:9999";
44     public static String topic = "test/dev";
45     public static int QoS = 1;
46     public static int timeout = 10000;
47     public static String clientId = "subscriber_1";
48     public static void main(String[] args) {
49         try {
50             MqttClient subscriber = new MqttClient(TCPAddress, clientId);
51             SubscriberCallbacks callbacks = new SubscriberCallbacks(clientId);
52             subscriber.setCallback(callbacks);
53
54             MqttConnectOptions options = new MqttConnectOptions();
55             options.setCleanSession(false);
56             options.setKeepAliveInterval(20);
57
58             System.out.println("Subscribing to topic \"" + topic +
59 "for instance " + subscriber.getClientId() + "with QoS = " + QoS);
60             subscriber.connect(options);
61
62             subscriber.subscribe(topic);
63             System.out.println("Press q to QUIT.");
64             Scanner scanner = new Scanner(System.in);
65             for (String input = ""; !input.equalsIgnoreCase("q");
66                 input = scanner.nextLine());
67
68             subscriber.disconnect();
69             System.out.println("Disconnected!");
70
71         } catch (Exception e) {
72             e.printStackTrace();
73         }
74     }
75 }
76 }
```

Bibliography

- [1] N. Kroes, G. Santucci, P. Friess et alii, *The Internet of Things - New Horizons*, Halifax UK, 2012
- [2] Yinghui Huang and Guanyu Li, *Descriptive Models for Internet of Things*, International Conference on Intelligent Control and Information Processing, 2010
- [3] Louise Coetzee and Johan Eksteen, *The Internet of Things - Promise for the Future? An Introduction*, IST-Africa Conference Proceedings, 2011
- [4] R. Coulouris, J. Dollimore, T. Kindberg and G. Blair, *Distributed Systems - Concept and Design*, Fifth Edition, Addison-Wesley, 2012
- [5] T. Kindberg., J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra and M. Spasojevic, *People, Places, Things: Web Presence for the Real World*, Mobile Networks and Applications 7, 365–376, 2002
- [6] D. Guinard, V. Trifa and E. Wilde, *A Resource Oriented Architecture for the Web of Things*, Proceedings of IoT 2010, IEEE International Conference on the Internet of Things. Tokyo, Japan, 2010
- [7] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess and D. Savio, *Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services*, IEEE Transactions on Services Computing, Vol. 3, No. 3, 2010
- [8] Building Facebook Messenger
<https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920>
- [9] MQTT official website
<http://www.mqtt.org>
- [10] Paho project wiki page
<http://wiki.eclipse.org/Paho>
- [11] MQ Telemetry Transport (MQTT) V3.1 Protocol Specification
http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/MQTT_V3.1_Protocol_Specific.pdf

- [12] Dave Evans, *The Internet of Things - How the Next Evolution of the Internet Is Changing Everything*, April 2011
http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf
- [13] Zach Shelby, *Introduction to Resource-Oriented Applications in Constrained Networks, Smart Object Tutorial*, IETF-80 Prague
<http://www.iab.org/wp-content/IAB-uploads/2011/04/Shelby.pdf>
- [14] Dave Locke, *Introduction to MQTT*, May 2013
<https://www.oasis-open.org/committees/download.php/49205/MQTT-OASIS-Webinar.pdf>
- [15] M. Stecca and M. Maresca, *An Execution Platform for Event Driven Mashups*, Proceeding iiWAS '09, Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services, pages 33-40, ACM, 2009

List of Figures

1.1	CoAP position in the ISO/OSI architecture.	3
1.2	REST API Design.	6
1.3	Xively platform overview.	8
1.4	Xively platform: adding a new development device.	9
1.5	Xively platform: device's resume screen.	10
1.6	Xively platform: batch adding.	10
2.1	Graphical user interface for creating mashups.	14
2.2	How a SP interacts with external world.	15
2.3	High level platform architecture.	16
2.4	Run time execution model.	17
3.1	An example of mashup.	19
3.2	The publish-subscribe paradigm	21
3.3	Message format.	23
3.4	Fixed header's bits structure.	23
3.5	Communication flow with QoS level 0	26
3.6	Communication flow with QoS level 1	27
3.7	Communication flow with QoS level 2	28
3.8	Paho project logo.	29
3.9	Terracotta Universal Messaging Architecture.	30
3.10	Universal Messaging Enterprise Manager.	31
3.11	Nirvana Manager Section	31
3.12	New Interface Section.	32
3.13	New interface parameters.	32
3.14	New channel form.	33
3.15	New channel icon on the left sidebar.	33
3.16	New channel permission form.	34
3.17	List of channel's permission.	34
4.1	System architecture with P/S broker.	37
4.2	An example of mashup.	39
4.3	Centralized SEP communication diagram.	43
4.4	Distributed SEP communication diagram.	50

List of Tables

1.1	Xivley service pricing table.	11
3.1	Possible message types.	24
3.2	Possible QoS levels.	25
4.1	Routing table for mashup in Figure 4.2.	39
4.2	Communication links tokens.	46
4.3	Loading comparison of centralized and distributed implementation.	51

Ringraziamenti

Desidero innanzitutto ringraziare il **prof. Carlo Ferrari** per l'attenzione dimostrata nei miei confronti, ed insieme a lui, l'**Ing. Michele Stecca** per avermi seguito e consigliato durante questo lavoro di tesi.

Alla **mia famiglia**, per tutti i sacrifici fatti per permettermi di arrivare dove sono. Grazie.

Ad **Elena**, per la pazienza portata durante questo periodo, per i giochi e gli scherzi insieme, per le scritte e i disegni sul quaderno, per il calendario countdown, per tutti i consigli di impaginazione, per la cena prima dei fiori, per gli abbracci nei momenti di panico, per i baci e gli sguardi di intesa... semplicemente perchè ci sei. Grazie!

A **Esterino e Carmela**, per i consigli di stile e avermi offerto rifugio i giorni prima della discussione. Grazie.

A **Ramo**, sei stata la prima persona che mi ha aperto la porta di Via Gorizia. Ricordo ancora il tuo "Ciaooo, Ramona!" mentre portavo su la mia valigia con il morto dentro. Con te in casa ci si poteva solo che divertire! Mi mancherai!

A **Chiara**, la mia bolzanina preferita. La regina del surgelato... Eri quasi riuscita a convertirmi con la caponata di Picard. Mi hai coccolato con il barattolo da quattro tonnellate di leibekuchen rendendomi l'uomo più felice della terra! Te lo dissi già dopo il tuo esame di stato, mi mancherai ma sono sicuro che rimarremo in contatto.

A **Luigi**, ci hai portato in casa una strage di donne! I nostri saluti mattutini passeranno agli annali.

Ad **Albe Z.**, mi hai iniziato alla buona festa padovana. Abbiamo condiviso molti Giovedì traumatici ma questo non ci ha fermato! Le partite a PES erano sempre un grande evento, per non parlare poi del poker del martedì!

A **Daniele**, per le chiacchierate davanti ad uno spritz. Il service ci sta dando grosse soddisfazioni. Andiamo avanti così!

A **Paola**, la donna dall'accento misto. Lo sai che sebbene ti prenda sempre in giro, in fondo, ti voglio bene!

A **Giulia**, la donna con lo spirito di bambina. Avevo già capito tutto a Torino con l'uva! Continua così...

A **Simone**, esimio collega. Per tutti gli sfoghi che ti sei dovuto sopportare ma soprattutto per essere stato lì ad ascoltarmi.

A **Federica**, la vagabonda della compagnia. Ti auguro il meglio per il tuo futuro.

A **Daniel**, il preparatore di pop corn pazzo. Continua ad istruirmi sulle ultime news calcistiche!!

Ad **Angelo**, il DJ che tutti i service ci invidiano! Come metti tu le canzoni, non le mette nessuno!! A parte gli scherzi, sei un ottimo collega...
Continuiamo così!

A **Frasca**, entità astratta, visibile in poche occasioni. Nonostante questo, ogni volta che ci vediamo è come se ci sentissimo ogni giorno.

Ad **Albino** e **Stefania**, per avermi accolto in casa vostra come un figlio.
Grazie!

A **Mosky**, il ragazzo super-impegnato. Spero di non aver impostato un livello troppo alto per il tuo finale di carriera universitaria! Sono sicuro che ne uscirai nel migliore dei modi.

A **Sofy**, mi diverto un sacco quando vengo a vederti giocare a calcetto. Spero di riuscire ancora a farlo!

A **Lory**, il presidente che ogni lega di fantacalcio vorrebbe avere. Gli eventi Apple sono più divertenti se visti in tua compagnia. AMAZING!

A **Berg**, mi stai alle calcagna in classifica. Sarà una sfida all'ultimo goal!

A **Giuggio**, per tutti i progetti che non ho mai avuto il tempo di seguire.
Prometto che almeno uno lo facciamo!!

A **Debbi**, per l'interessamento mostrato nei miei confronti. Buon percorso!

A **Bedo**, fidato compagno di gruppo durante i corsi che ci hanno reso un po' più imprenditori. Tieni a portata di mano la tua reflex che ci sono un sacco di eventi da fotografare.

A **Baruz**, per avermi sempre spinto a tenere un monologo sullo skeumorphismo. Potrei usarlo nei colloqui di lavoro!

A **Ale**, egregio compagno di avventure durante Computer Networks Managemet. Quando ci ricapiterà un esame così?

A **Cecca**, fantastico tanguero e supporto morale durante le lezioni di tango.
Però, in fin dei conti, mi sono piaciute!

A **Biss**, mi hai aiutato un sacco durante il progetto di Dati 3D, sopportando mie uscite a caso sulle registrazioni. Alla fine, comunque, anche tu hai vinto contro il Comau!

A **Fabio B.**, con la tua *beccanea* ho riso un sacco. Ti auguro ogni bene per il tuo futuro.

A **Fabio G.**, principale esperto della filmografia di Maccio Capatonda.
Trascorrere le pause ripassando tutti i trailer era sicuramente divertente.

A **Filippo**, con te le risate sono assicurate. Ti auguro di concludere la tua esperienza al DEI nel migliore dei modi.

Al **Signor P.**, per avermi fatto compagnia durante le sessioni di programmazione pazza e disperata.

A tutti quelli che, in qualunque modo, hanno fatto parte di questo viaggio durato cinque anni.