Connecting ROS and FIWARE: concepts and tutorial

Raffaele Limosani, Alessandro Manzi, Laura Fiorini,
Paolo Dario, and Filippo Cavallo

The BioRobotics Institute, Scuola Superiore Sant'Anna,
Viale Rinaldo Piaggio, 34, 56026 Pontedera (PI), Italy
r.limosani@santannapisa.it

https://www.santannapisa.it/en/institute/biorobotics/biorobotics-institute

Abstract. Nowadays, the Cloud technology permeates our daily life, spread in various services and applications used by modern instruments, such as smartphones, computer, and IoT devices. Besides, the robotic field represents one of the future emerging markets. Nevertheless, these two distinct worlds seem to be very far from each other, due to the lack of common strategies and standards.
The aim of this tutorial chapter is to provide a walkthrough to build a basic Cloud Robotics application using ROS and the FIWARE Cloud framework. At the beginning, the chapter offers step-by-step instructions to create and manage an Orion Context Broker running on a virtual machine. Then, the `firos` package is used to integrate the ROS topic communication using publishers and subscribers, providing a clear example. Finally, a more concrete use case is detailed, developing a Cloud Robotics application to control a ROS-based robot through the FIWARE framework.
The code of the present tutorial is available at `https://github.com/Raffa87/ROS_FIWARE_Tutorial`, tested using ROS Indigo.

Keywords: Tutorial ROS FIWARE Firos

1 Introduction

Nowadays, the ROS framework [1] is considered the "de facto" robotic standard in academic research and it is widely adopted all over the world [2]. Moreover, in the last years, the ROS-Industrial project is trying to extend the advanced capabilities of ROS also to manufacturing automation. Therefore, it is reasonable to forecast a much intense use of ROS in the near future for developing complex services involving several interacting robots. However, the ROS framework is conceived to develop systems that run on machines connected to the same LAN network. To overcome this limitation, different attempts have been made to enhance the ROS capabilities through the use of Cloud resources, leveraging the so-called "Cloud Robotics". One of the definitions of this concept is "any

robot or automation system that relies on either data or code from a network to support its operation, i.e., where not all sensing, computation, and memory are integrated into a single standalone system" [3].

Several research groups have focused their efforts on cloud robotic challenges: for example, in [4] authors extended the computation and information sharing capabilities of networked robotics by proposing a cloud robotic architecture, while in [5] the key research issues in cloud network robotics are discussed analyzing a case study in a shopping mall. One of the first application of the Cloud Robotics paradigm is the DAvinCi [6] project developed in 2010, which focus more on the computational side rather than in communication between robot and server. Another example is represented by Rapyuta [7], an open-source Cloud Robotics platform, developed during the RoboEarth project [8]. Nevertheless, the overall performances do not differ much from other solutions [9]. Furthermore, cloud infrastructures have been developed to teach ROS[1], focusing on the release of easy-to-use development instruments.

Despite the aforementioned researches, robotics and cloud technologies still appear two distinct worlds in terms of standards, communities, instruments, and infrastructures. Although the robotic field represents an emerging future market, the Cloud is currently an active business. In fact, most of the big Hi-Tech companies offer services over the cloud, in the form of Software-as-a-Service, or Platform-as-a-Service, including the Google Cloud Platform[2], the Amazon Web Services[3] and the Microsoft Azure[4].

The importance of the Cloud technology is also underlined by the European Union, which founded the universAAL project[5][10], and, more recently, the FIWARE[6] [11] program, which provides a middleware platform for the development of applications for Future Internet.

FIWARE provides an enhanced OpenStack-based cloud environment plus a rich set of open standard APIs that make it easier to connect to the Internet of Things, process and analyze Big data and real-time media or incorporate advanced features for user interaction.

Even if a link between ROS and FIWARE has been already implemented[7][12], its use is still difficult due to a lack of tutorials and documentation for users with different backgrounds (i.e. experts in ROS without previous experience in FIWARE or vice-versa).

The aim of this tutorial chapter is to provide a step-by-step walkthrough about the integration of ROS with FIWARE, starting from the creation of a basic virtual machine over the Cloud. It shows how to handle the communication using subscribers and publishers introducing the `firos` package. At the end, a

[1] `http://www.theconstructsim.com/`, visited on December 2017
[2] `https://cloud.google.com/`, visited on December 2017
[3] `https://aws.amazon.com/`, visited on December 2017
[4] `https://azure.microsoft.com`, visited on December 2017
[5] `http://universaal.aaloa.org/`, visited on December 2017
[6] `https://www.fiware.org`, visited on December 2017
[7] `http://wiki.ros.org/firos`, visited on December 2017

concrete Cloud Robotics example is provided to control a simulated robot using the FIWARE framework.

In details, this chapter covers the following aspects:

- basic concepts about ROS and FIWARE;
- introduction to FIWARE and the `firos` package;
- creation and management of a FIWARE Context Broker;
- how to use the FIWARE Context Broker;
- how to connect ROS and FIWARE;
- how to develop a robot control application through FIWARE.

At the end of the tutorial, despite the initial background, the reader will be able to manage communication among ROS and FIWARE, allowing to enhance robotic projects with Cloud capabilities. The final aim is to provide instruments and documentation that will improve the collaborations between the robotic and the Cloud worlds.

The remainder of this tutorial chapter is organized as follows. The basic concepts of the ROS middleware and the FIWARE framework are provided in Section 2 and in Section 3 respectively. A walkthrough about the creation and the use of the FIWARE Orion Context Broker and *firos* aare detailed in Section 4 and 5, showing a first "*Hello world*" example. A more complex and concrete use case is described in Section 6, explaining how to develop a basic Cloud Robotics application using FIWARE to send a goal and receive data from a simulated robot. Finally, the Section 7 concludes the chapter and discusses further developments.

2 ROS concepts

ROS[8] is an open-source, meta-operating system for robot control and development. The term "meta-operating" is used to indicate its similarity with an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality and message-passing between processes. On the other hand, ROS is not an operating system in a strict sense; it currently runs on Unix-based platforms.

As the main feature, ROS provides a robust infrastructure that simplifies communications among processes. The ROS runtime "graph" is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using a common communication infrastructure. ROS implements different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and data storage through the Parameter Server.

[8] More information about ROS can be found at `http://wiki.ros.org/ROS`, visited on December 2017.

2.1 ROS Graph

The Computation Graph[9] is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways.

For the purpose of this tutorial, only details about nodes, Master, messages, and topics are reported.

Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one node provides a graphical view of the system, and so on.

Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other or exchange messages.

Messages are used by the nodes to communicate with each other. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays.

Topics Messages are routed via a transport system with publish/subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.

The Master acts as a nameservice in the ROS Computation Graph. Nodes connect to other nodes directly; the Master only provides lookup information, much like a DNS server, allowing nodes to dynamically create connections as new nodes are run.

Refer to the official ROS website[10] for details about the framework installation.

[9] Deeper information on ROS Graph can be found at `http://wiki.ros.org/ROS/Concepts`, visited on December 2017.

[10] `http://wiki.ros.org/ROS/Installation`, visited on December 2017.

3 FIWARE concepts

The *FIWARE platform*[11] provides a rather simple yet powerful set of APIs that ease the development of smart applications in multiple vertical sectors. The specifications of these APIs are public and royalty-free. Besides, an open source reference implementation of each of the FIWARE components is publicly available so that multiple FIWARE providers can emerge faster in the market with a low-cost proposition.

For the purpose of this tutorial, reader will be leaded to be a user of FIWARE Lab, which is a non-commercial sandbox environment where innovation and experimentation based on FIWARE technologies take place. Entrepreneurs and individuals can test the technology as well as their applications, exploiting Open Data published by cities and other organizations. FIWARE Lab is deployed over a geographically distributed network of federated nodes leveraging on a wide range of experimental infrastructures.

In details, the FIWARE framework contains a rich library of components, called Generic Enablers (GE), that allow developers to put into effect functionalities such as the connection to the Internet of Things or Big Data analysis. By combining them, it is possible to develop modular and complex applications.

The GEs offer a number of general-purpose functions, including:

Data/Context Management to easy access, gather, process, publish, and analyze context information (e.g. Comet, Cygnus, Kurento, Orion).

IoT Services to use, search, and access IoT devices and sensors, handling various protocols.

Web-based User Interface to give tools for 2D/3D graphics, and Geographical Information System (GIS) Data Provider.

Security to implement security and privacy requirements (e.g. PEP Proxy, KeyRock, AuthZForce).

Cloud Hosting to provide computation, storage, and network resources (e.g. Docker, Murano, Bosun, Pegasus).

The deployment of a GE is referred as an *instance*. FIWARE allows to fully customize new instances or use a set of pre-configured instances available in the framework. For the purpose of this tutorial walkthrough, the remainder will focus on the Orion Context Broker, which is one of the key component of a FIWARE application. In addition, this section also introduces `firos`, which allows to connect a ROS node with Orion.

3.1 The Orion Context Broker

The Orion Context Broker[12] is a C++ implementation of the NGSIv2 REST API binding developed as a part of the FIWARE platform. The NGSI specification is an information model that uses the concept of entities to virtually represent

[11] `https://www.fiware.org`, visited on December 2017.

[12] `http://fiware-orion.readthedocs.io/en/master/index.html`

physical objects in the real world. Any information about physical entities is expressed in the form of attributes of virtual entities[13].

The Orion Context Broker allows to manage all the whole lifecycle of context information including updates, queries, registrations and subscriptions. It implements an NGSIv2 server to manage context information and its availability. Using the Orion Context Broker, the user is able to create context elements and manage them through updates and queries. In addition, it is possible to subscribe to context information and automatically receive a notification when some condition occurs (e.g. a context element has changed).

3.2 Firos

`firos` [12] represents the link between ROS and FIWARE. In particular, it is a ROS node that communicates with the Orion Context Broker to publish and listen robot data. In other words, `firos` works as a translator between the robotics field and the cloud world, transforming ROS messages into NGSI to publish them in the cloud, and vice-versa.

4 Walkthrough: How to Use the Orion Context Broker

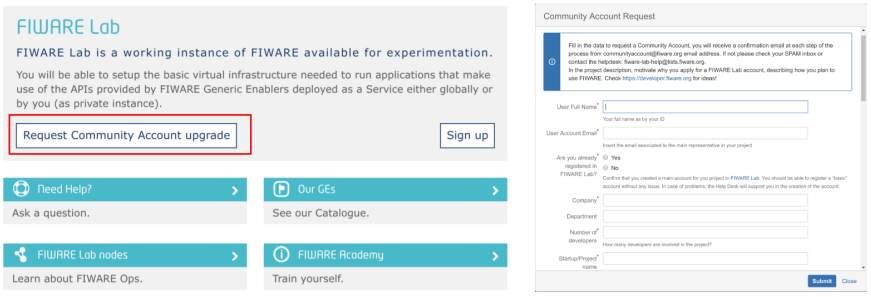
As introduced, after a short description of ROS and FIWARE concepts, the aim of the paper is to provide working tutorial where the reader can understand new concepts and tools through practical example. Three walkthroughs are presented, in a sort of growing difficulty:

1. in the first one, the use of Orion Context Broker as "container" of data is explained, after a detailed step-by-step guide on the creation in FIWARE platform;
2. in the second one, an "Hello World" example is depicted showing the mechanism of communication-based on `firos` ROS node;
3. in the third one, a more concrete example is presented showing how FIWARE and ROS can be used to create a cloud-robotic application where commands (e.g. navigation goals) can be remotely sent by a third party and feedbacks (e.g. odometry) can be remotely received.

4.1 First step: create a FIWARE Lab account

The first step to begin the walkthrough is to become a FIWARE Lab user, therefore create an account at `https://account.lab.fiware.org`. Mark the option "*I want to be a trial user*". This procedure will create a trial account lasting 14 days. Check the provided e-mail and confirm the account. To proceed with the tutorial, we need to upgrade the trial account to be able to instantiate the required GE. Hence, log in, go to the account settings and click on *Account Status*. Here it is possible to request a *Community Account upgrade*.

[13] More information can be found at `http://aeronbroker.github.io/Aeron`



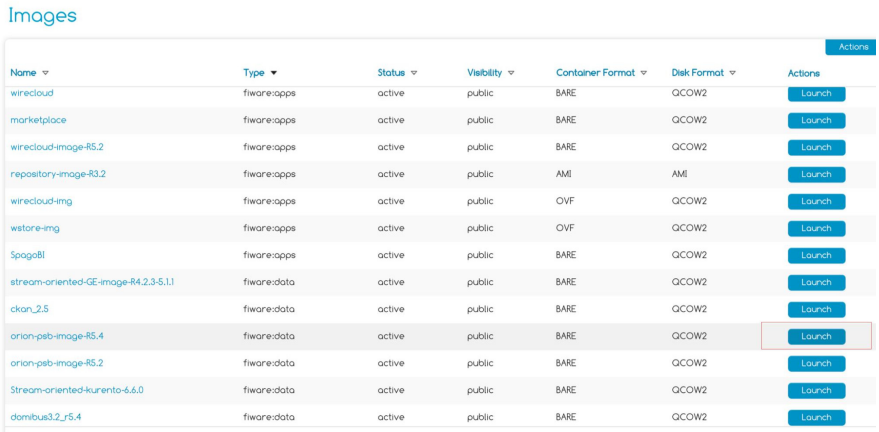
For the purpose of this tutorial, the reader can select the *default quota* on the relative upgrade form. It is worth to mention that the account upgrade process is not immediate and can require up to two days.

4.2 Create a Orion instance

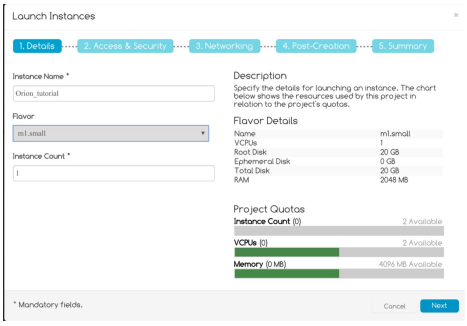
This section covers the creation of a Orion Context Broker instance. The FIWARE catalog contains a set of pre-configured instances to easy deploy specific Virtual Machine (VM) on the Cloud. The use of pre-configured instances is not mandatory and the user can also manually create its own personalized instance. The present tutorial will use a pre-configured Orion instance, which automatically creates a Virtual Machine (VM) equipped with the Context Broker running on a centOS machine.

Create the instance To create the new Orion instance, move to the *Cloud* tab on the FIWARE GUI and click on the *Launch New Instance*.

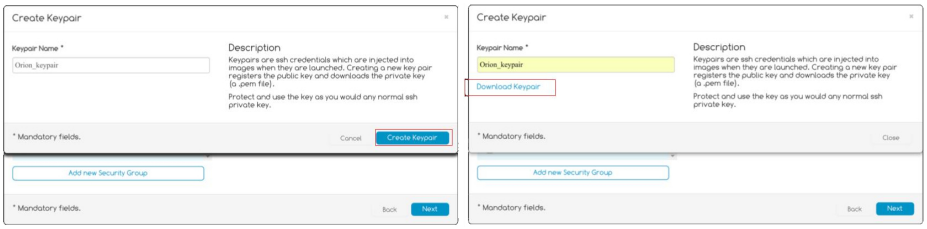
Choose an Orion Context Broker image from the list (*orion-psb-image-R5.4*) and launch it.



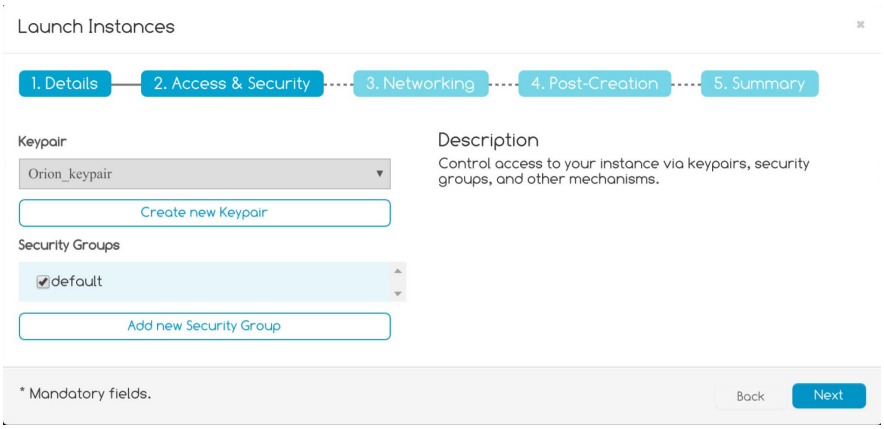
After that, a *Launch Instances* window related to its settings will appear. First of all, provide the instance *details*, choosing a name and selecting its `flavor` (i.e. resources of the VM). For the aim of this tutorial, it is enough to choose the `m1.small` option, which allocates a VM with 1 CPU, 2Gb RAM, and 20 Gb of disk space:



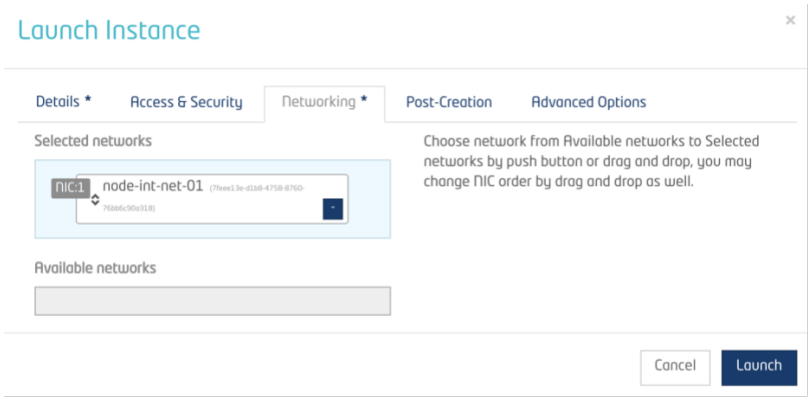
After that, you can switch to the *Access & Security* tab to generate the key-pairs needed for the secure SSH remote access to the instance. By clicking on `Create Keypair` the VM will be configured for the remote connection and the relative key-pair will be download from the host computer:



Use the new created keypair and check the `default` option in the *Security Group* field:

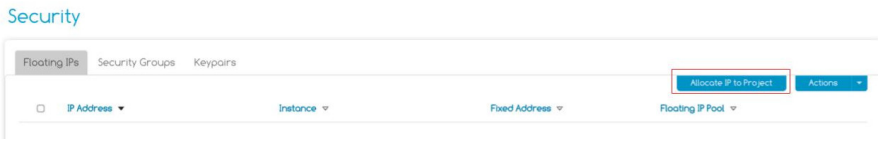


Afterwards, use the *Networking* tab to assign a network by moving the created instance from `Available Networks` to `Selected Networks`:

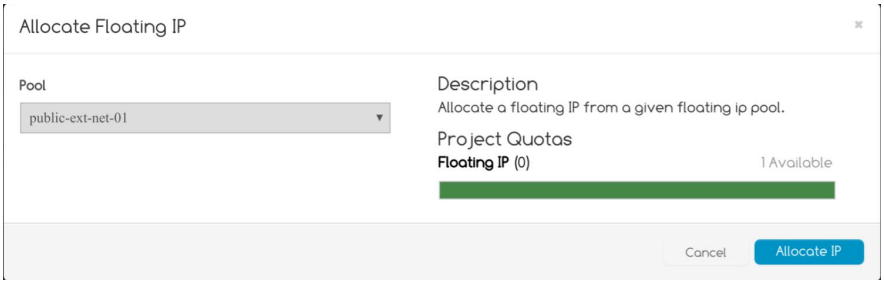


Eventually, continue through the final steps *Post-Creation* and *Summary* to launch the new instance. For the purpose of the tutorial, in these steps is not needed any additional information.

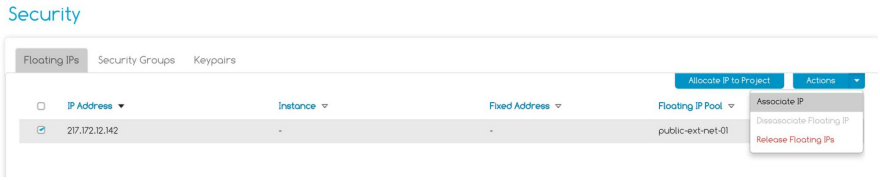
Connecting to the instance To remotely connect to the newly created instance, first a public IP must be assigned. Therefore, move to the `Security` tab and click on `Allocate IP to Project button`:



Now, use the default `public-ext-net-01` option for `Pool` and click on `Allocate IP`:



Finally, you can associate the IP with the instance selecting it and clicking on `Actions: Associate IP`



After the confirmation, the created VM is accessible from the Internet. In the remainder, we provide the detailed instruction to connect to the instance using an SSH client for Linux-based operating system[14].

1. Open the Terminal
2. Locate the key-pair associated to the instance when launching it (e.g. `Orion_keypair.pem`)
3. Modify the key-pair permissions to make it not publicly viewable:

```
$ chmod 400 Orion_keypair.pem
```

4. Connect to the instance using its public IP:

```
$ ssh -i Orion_keypair.pem username@public_ip
```

Now, you have the remote access to the instance, and in the next, the tutorial will cover the necessary steps on how to use the Orion Context Broker.

4.3 Orion Context Broker Example Case

Before discussing how to integrate FIWARE with robotics, let's introduce an Orion example case[15] to fully understand its capabilities.

This section will show how to manage the context information of a building with several rooms using the Orion Context Broker. The rooms are `Room1`, and `Room2` equipped with two sensors: temperature and (atmospheric) pressure.

The Orion Context Broker interacts with a *context producer* applications (which provide sensor information) and a *context consumer* application (which processes that information, e.g. to show it in a graphical user interface). We will show how to act both as producer and consumer.

Starting the broker Start the broker (as root or using the `sudo` command) on the FIWARE instance:

```
$ sudo /etc/init.d/contextBroker start
```

On normal behavior the output is:

[14] Windows user can use the PuTTY client or similar, please refer to the relative documentation.

[15] Taken from `http://fiware-orion.readthedocs.io/en/master/user/walkthrough_apiv2/index.html`

```
Starting...
contextBroker (pid 1372) is running...
```

In case you need to restart the broker, execute the following:

```
$ sudo /etc/init.d/contextBroker restart
```

Issuing commands to the broker To issue requests to the broker, this tutorial uses the `curl` command line tool, because it is often present in any GNU/Linux system and simplifies the explanation. Of course, it is not mandatory, and the reader can use any REST client tool instead (e.g. RESTClient). Indeed, in a real case, a developer will probably interact with the Orion Context Broker using a programming language library implementing the REST client.

The basic patterns for all the `curl` examples in this document are the following:

POST

```
curl localhost:1026/<operation_url> -s -S [headers]' -d @- <<EOF
[payload]
EOF
```

PUT

```
curl localhost:1026/<operation_url> -s -S [headers] -X PUT -d @- <<EOF
[payload]
EOF
```

PATCH

```
curl localhost:1026/<operation_url> -s -S [headers] -X PATCH -d @- <<EOF
[payload]
EOF
```

GET

```
curl localhost:1026/<operation_url> -s -S [headers]
```

DELETE

```
curl localhost:1026/<operation_url> -s -S [headers] -X DELETE
```

Regarding the `headers`, it is possible to include the following ones:

Accept header, to specify which payload format you want to receive in the response. User should explicitly specify JSON:

```
curl ... --header 'Accept: application/json' ...
```

Content-Type header, when a payload is needed for requests (i.e. POST, PUT or PATCH). Also in this case, user should explicitly specify JSON:

```
curl ... --header 'Content-Type: application/json' ...
```

Regarding the aforementioned commands, please take into account the following remarks:

- most of the time we are using multi-line shell commands, using `EOF` to mark the beginning and the end of the multi-line block;
- in some cases (GET and DELETE), we omit `-d @-` as they don't use payload;
- we assume that the broker is listening on port 1026. Adjust this in the `curl` command line in case of different setting;
- to pretty-print JSON in responses, we use the `json.tool` from the json Python module:

```
(curl ... | python -m json.tool) <<EOF
...
EOF
```

Context Management At this step, we are ready to create both producer and consumer applications, using the Orion Context Broker. At the beginning, the broker starts in an empty state. Therefore, we need to make it aware of the existence of certain entities. In particular, we are going to create the entities for `Room1` and `Room2`, each one with two attributes (temperature and pressure). We create the entities using the `POST /v2/entities` operation. First of all, we declare the `Room1` entity with 23°C and 720 mmHg of temperature and pressure respectively:

```
curl localhost:1026/v2/entities -s -S --header 'Content-Type: \
application/json' -d @- <<EOF
{
  "id": "Room1",
  "type": "Room",
  "temperature": {
    "value": 23,
    "type": "Float"
  },
  "pressure": {
    "value": 720,
    "type": "Integer"
  }
}
EOF
```

The entity is defined with an `id` and a `type`, and contains 2 attributes with a `value` and a `type` field. It is important to know that the Orion Context Broker does not perform any type check. In other words, it will accept a temperature value either if it is formatted as a float like 25.5 or as a string like *hot*. This means that the developer has to take care of the correctness of the data types

during the implementation. Upon receipt of this request, the broker will create the entity in its internal database, it will set the values for its attributes and it will respond with a 201 Created HTTP code.

In the same way, we create the `Room2` entity, setting temperature and pressure to 21°C and 711 mmHg respectively:

```
curl localhost:1026/v2/entities -s -S --header 'Content-Type: \
application/json' -d @- <<EOF
{
  "id": "Room2",
  "type": "Room",
  "temperature": {
    "value": 21,
    "type": "Float"
  },
  "pressure": {
    "value": 711,
    "type": "Integer"
  }
}
EOF
```

Now, we can act as a consumer application, wanting to access the context information stored by the Orion Context Broker to do something interesting with it (e.g. show a graph with the room temperature in a graphical user interface). In this case, we use the `GET /v2/entities/{id}` request. To retrieve the context information of the `Room1` use the following command:

```
curl localhost:1026/v2/entities/Room1?type=Room -s -S \
    --header 'Accept: application/json' | python -m json.tool
```

Actually, you don't need to specify the type, as in this case there is no ambiguity using just the ID, so you can also do:

```
curl localhost:1026/v2/entities/Room1 -s -S \
    --header 'Accept: application/json' | python -m json.tool
```

In both cases, the response includes all the attributes belonging to `Room1`:

```
{
    "id": "Room1",
    "pressure": {
        "metadata": {},
        "type": "Integer",
        "value": 720
    },
    "temperature": {
        "metadata": {},
        "type": "Float",
        "value": 23
```

```
    },
    "type": "Room"
}
```

It is also possible to use the `keyValues` option in the request:

```
curl localhost:1026/v2/entities/Room1?options=keyValues -s -S \
    --header 'Accept: application/json' | python -m json.tool
```

which produces a compact response, including just the attribute values:

```
{
    "id": "Room1",
    "pressure": 720,
    "temperature": 23,
    "type": "Room"
}
```

A third way consists in requesting specific attributes using the `values` option plus a list of attribute name. To do so, use the `attrs` URL parameter to specify the order. For instance, using this command (temperature first, pressure second):

```
curl 'localhost:1026/v2/entities/Room1?options=values&attrs=temperature,
    pressure' -s -S \
    --header 'Accept: application/json' | python -m json.tool
```

produce the following response:

```
[
    23,
    720
]
```

Finally, note that requesting a non-existing entity will produce the following error:

```
{
    "description": "The requested entity has not been found. \
       Check type and id",
    "error": "NotFound"
}
```

The same happens for non-existing attribute:

```
{
    "description": "The entity does not have such an attribute",
    "error": "NotFound"
}
```

Another useful way to get all the context information is to list all the entities omitting the `id` in the `GET` command:

```
curl localhost:1026/v2/entities -s -S \
    --header 'Accept: application/json' | python -m json.tool
```

which, in our case, return both `Room1` and `Room2`:

```
[
    {
        "id": "Room1",
        "pressure": {
            "metadata": {},
            "type": "Integer",
            "value": 720
        },
        "temperature": {
            "metadata": {},
            "type": "Float",
            "value": 23
        },
        "type": "Room"
    },
    {
        "id": "Room2",
        "pressure": {
            "metadata": {},
            "type": "Integer",
            "value": 711
        },
        "temperature": {
            "metadata": {},
            "type": "Float",
            "value": 21
        },
        "type": "Room"
    }
]
```

At this step, we are able to produce and consume information using the Orion Context Broker. In the next, this tutorial will focus on the integration between ROS and FIWARE.

5 Walkthrough: Connecting ROS and FIWARE

The link between ROS and FIWARE is represented by the `firos` package [12]. For the aim of this tutorial, we refer to our fork version of the original package available at `https://github.com/Raffa87/ROS_FIWARE_Tutorial`[16]. The forked package contains a simplified version of the code with utilities already set

[16] The original `firos` package is available at `https://github.com/Ikergune/firos`

for the present tutorial. At the end, the reader will be able to send and receive data using ROS topics and the FIWARE Orion Context Broker.

5.1 Install Firos

First of all, clone the repository into your ROS workspace

```
$ git clone https://github.com/Raffa87/ROS_FIWARE_Tutorial
```

build the package

```
$ catkin_make
```

The `firos` package is now installed and needs to be configured according to the specific application.

5.2 Configuring FIROS

The configuration files are located at the `src/firos/config` folder. Open the `config.json` file and fill the following parameters:

address: the public IP of the FIWARE instance (e.g. 217.123.12.123)
port: the port used by the Orion Context Broker (default: 1026)
interface: the network interface used by the PC (e.g. `wlan0`, `eth0`)

Open the file *robots.json* to declare the ROS topics that will be used by `firos`. For instance, using the following configuration file:

```
{
  "end_end_test":{
       "topics": {
           "p1": {
               "msg": "std_msgs.msg.String",
               "type": "publisher"
           },
           "s1": {
               "msg": "std_msgs.msg.String",
               "type": "subscriber"
           }
       }
}
```

will configure `firos` to transmit the data received on the `s1` topic to the specified Orion Context Broker and to republish on the corresponding topic the change of information on `p1` of `end_end_test` entity.

5.3 Run FIROS

To execute the `firos` node:

```
$ rosrun firos core.py
```

The output of a running node with the previous configurations will be like the following:

```
Initializing ROS node: firos
Initialized
Starting Firos setup...
--------------------------------
Generating Message Description Files
Successfully generated
Starting Firos...
--------------------------------
Getting configuration data
Generating topic handlers:
  -end_end_test
    -p1
    -s1
Subscribing on context broker to ROBOT end_end_test and topics: []
Connected to Context Broker with id 59ba8a5bb05d744325f22525
Subscribed to end_end_test's topics

Press Ctrl+C to Exit

Serving HTTP on 0.0.0.0 port 10100 ...
```

5.4 From ROS topic to Orion Context Broker

At this step, we can check the connection between the `firos` node and the Orion Context Broker by publishing on the specified `s1` topic:

```
$ rostopic pub /s1 std_msgs/String "hello world" __ns:=end_end_test
```

After that, the `firos` node modifies the attribute of the entity `end_end_test` in the Orion Context Broker. We can verify it by querying Orion in the following way:

```
curl contextbroker_ip:1026/v2/entities/end_end_test -s -S \
  --header 'Accept: application/json' | python -mjson.tool
```

The response will look like as:

```
{
    "COMMAND": {
        "metadata": {},
        "type": "COMMAND",
        "value": ""
    },
    "id": "end_end_test",
    "s1": {
```

```
        "metadata": {},
        "type": "std_msgs.msg.String",
        "value": "{%27firosstamp%27: 1505398290.893342,
            %27data%27: %27hello world%27}"
    },
    "type": "ROBOT"
}
```

As we can see from the response, the communication between ROS and FIWARE was successful. In particular, the use and configuration of `firos` automatically create a new entity of type `ROBOT` that has an attribute named as the subscribed topic whose value is automatically set and updated according to the data flowing on the topic.

5.5 From Orion Context Broker to ROS topic

In a similar way, we can use `firos` to gather data from the context broker to ROS topics. The *robots.json* configuration file already specifies a subscriber for the `p1` topic in the `end_end_test` namespace. However, although the `firos` node automatically creates subscriber attributes for the specified entity, it does not do the same for publishers. Therefore, we have to add a publisher attribute to our entity in the Context Broker:

```
curl contextbroker_ip:1026/v2/entities/end_end_test/attrs -s -S \
    --header 'Content-Type: application/json' -d @- <<EOF
{
    "p1": {
        "metadata": {},
        "type": "std_msgs.msg.String",
        "value": "test"
    }
}
EOF
```

Now, to verify the communication, open another terminal on the machine where `firos` is running and echo the topic:

```
rostopic echo /end_end_test/p1
```

After that, modify the value of `p1` attribute on the Orion Context Broker using the `PATCH` command:

```
curl contextbroker_ip:1026/v2/entities/end_end_test/attrs -s -S \
    --header 'Content-Type: application/json' -X PATCH -d @- <<EOF
{
    "p1": {
             "metadata": {},
             "type": "std_msgs.msg.String",
             "value": "{%27data%27: %27`echo $RANDOM`%27}"
```

```
   },

    "COMMAND": {
           "type": "COMMAND",
           "value": ["p1"]
     }
}
EOF
```

The new value will be printed in the `rostopic echo` terminal. At this point, we have demonstrated how to connect ROS with FIWARE, both on subscription and publication side. However, concerning the subscription case, it is worth to note that the Orion Context Broker has to be able to reach the IP address of the machine running the `firos` node. In other words, the robot has to be on the same network of the cloud resource. This, of course, represents a huge limitation, and the remainder will face this visibility issue.

6 Walkthrough: How To control a Robot through FIWARE

So far we have seen how to practically use the FIWARE Context Broker (Section 4) and how to use the `firos` package to simply integrate FIWARE with ROS using publishers and subscribers (Section 5). In this section, we go a step forward providing a more concrete Cloud Robotics example. We will see how to send a goal command to a robot, and look at the odometry data passing through the FIWARE Orion instance. To be close to a real Cloud Robotics application, we propose a system in which the control station is not in the same network of the FIWARE cloud resource (i.e. we do not setup a dedicated VPN). To solve the visibility issue between different machines over the Internet, we implement a specific module on the robot that periodically asks for commands to execute (polling-mode).

In details, the following step will be implemented and explained:

- create a new entity in the Orion Context Broker to store new commands for a mobile platform through the `PUT` and `PATCH` commands;
- use Gazebo to simulate a Clearpath[17] Husky mobile platform equipped with a navigation stack to move in the environment;
- use `firos` to update odometry information in the Orion Context Broker;
- implement a python script to periodically poll new command from the Orion Context Broker through the `GET` command;
- retrieve updated odometry data from the Orion Context Broker through the `GET` command.

In the following, we provide a detailed walkthrough that details the proposed system.

[17] `https://www.clearpathrobotics.com`, visited on December 2017.

6.1 Create a new entity to store robot commands

First of all, let's create a new bare entity on the Orion Context Broker where to store the commands for the robot:

```
curl contextbroker_ip:1026/v2/entities -s -S \
   --header 'Content-Type: application/json' -d @- <<EOF
{
  "id": "Husky_fiware_command",
  "type": "Robot_command",
  "command": {
    "value": "test",
    "type": "String"
  }
}
EOF
```

We have created a `Husky_fiware_command` entity having a `command` attribute which is defined through a `value` and `type` field. We will see later, how the `value` field will contain the information for a goal command, formatted as a ROS message, while the latter will contain the ROS message type.

6.2 Simulate a Husky mobile platform

In our example, we will use a simulated Husky platform using the Gazebo software. We decided to use it because it is well documented[18] and easy to reproduce. Hence, for simulating the robot we use the `husky_gazebo` and `husky_navigation` packages. In this walkthrough, we use the `husky_fiware` namespace, so our tutorial repository, already contains the launch files for those packages that refer the ROS topics to our specific namespace. Therefore, start the pre-configured Husky simulation environment where `ns` is already specified, also as argument of the `gazebo_ros/spawn` node to keep consistency between simulated robot and running ROS nodes.

```
$ roslaunch fiware_polling_command husky_playpen_fiware_demo.launch
```

and the `husky_navigation` stack in the `husky_fiware` namespace:

```
$ roslaunch husky_navigation move_base_mapless_demo.launch __ns:=husky_fiware
```

The Gazebo simulator will open showing a Husky robot ready to execute commands.

The rationale beyond the use of namespace is related to the `firos` implementation: `firos` node could be used to simultaneously manage several robots: for each robot, a different entity and topics are published and subscribed using the specified namespace.

[18] `http://wiki.ros.org/husky_navigation/Tutorials`, visited on December 2017.

6.3 Update the odometry on Orion Context Broker

To update the odometry information on the Orion Context Broker, we have to change the `firos` config file, as we did in Section 5.2. Hence, modify the `robots.json` file to specify that the `odometry/filtered` topic has to be forwarded to the Orion Context Broker. The modified file looks like:

```
{
  "end_end_test":{
       "topics": {
           "p1": {
               "msg": "std_msgs.msg.String",
               "type": "publisher"
           },
           "s1": {
               "msg": "std_msgs.msg.String",
               "type": "subscriber"
           }
       }
    },
  "husky_fiware":{
       "topics": {
           "odometry/filtered": {
               "msg": "nav_msgs.msg.Odometry",
               "type": "subscriber"
           }
       }
    }
}
```

We declare a subscriber on the `odometry/filtered` topic expecting a ROS `nav_msgs.msg.Odometry` message. Now, you can run the `firos` node:

```
$ rosrun firos core.py
```

To complete the robot side programs, run the `fiware_polling_command.py` script that periodically request (through a GET) the value of the `command` attribute of the `Husky_fiware_command` entity. The script also parses the command, publishing it on the relative navigation stack topic. The code of the script is the following:

```
#!/usr/bin/env python

import time
import requests
import json
import rospy
from move_base_msgs.msg import MoveBaseGoal
from geometry_msgs.msg import PoseStamped
```

```
10 def fiware_polling():
       context_broker_ip = IP_ADDRESS
12     pub = rospy.Publisher('/husky_fiware/move_base_simple/goal',
           PoseStamped, queue_size=10)
14     rospy.init_node('fiware_polling_goal', anonymous=True)
       rate = rospy.Rate(1) #1hz
16     while not rospy.is_shutdown():
        headers = {
18          'Accept': 'application/json',
        }
20      r = requests.get('http://' + context_broker_ip +
            ':1026/v2/entities/Husky_fiware_command', headers=headers)
22      j = json.loads(r.text)
        if ("PoseStamped" in j['command']['type']):
24       try:
          msg = PoseStamped()
26        msg.header.frame_id = j['command']['value']['header']['frame_id']
          msg.pose.position.x = j['command']['value']['pose']['position']['x']
28        msg.pose.position.y = j['command']['value']['pose']['position']['y']
          msg.pose.position.z = j['command']['value']['pose']['position']['z']
30        msg.pose.orientation.x = j['command']['value']['pose']['orientation']['x']
          msg.pose.orientation.y = j['command']['value']['pose']['orientation']['y']
32        msg.pose.orientation.z = j['command']['value']['pose']['orientation']['z']
          msg.pose.orientation.w = j['command']['value']['pose']['orientation']['w']
34        pub.publish(msg)
          rate.sleep()
36       except TypeError:
          continue
38
        headers = {
40          'Content-Type': 'application/json',
        }
42      data = '{"command": {"value": "none","type": "String"}}'
        r = requests.patch('http://' + context_broker_ip + ':1026/v2/entities/
44          Husky_fiware_command/attrs', headers=headers, data=data)

46 if __name__ == '__main__':
       try:
48       fiware_polling()
       except rospy.ROSInterruptException:
50       pass
```

In details, it implements a ROS node, which instantiate a publisher for the navigation stack of the robot (line 12). Then, it continuously asks the Context Broker about the `command` attribute through a `GET` request (line 20). Then, it

checks if the type field is equal to the expected `PoseStamped` (line 23). After that, it parses the retrieved value and fills a new `geometry_msgs/PoseStamped` message that is published on the `/husky_fiware/move_base_simple/goal` topic (lines 25-34). At the end, it reset the `command` attribute on the Orion Context Broker using `PATCH` (line 43).

6.4 Store robot commands on the Context Broker

New commands can be set by a third-party application (e.g. user interface) through a PATCH message, modifying the value of the `command` attribute of the `Husky_fiware_command` entity. A possible message looks like the following:

```
curl contextbroker_ip:1026/v2/entities/Husky_fiware_command/attrs -s -S \
   --header 'Content-Type: application/json' \
     -X PATCH -d @- <<EOF
{
    "command": {
        "value": {
              "header" : {
                   "seq" : 1,
                   "time" : 0.0,
                   "frame_id" : "base_link"
              },
              "pose" : {
                  "position" :
                   {
                     "x" : 0.0,
                     "y" : -1.0,
                     "z" : 0.0
                   },
                  "orientation" :
                   {
                      "x" : 0.0,
                      "y" : 0.0,
                      "z" : 0.0,
                      "w" : 1.0
                   }
               }
        },
        "type": "PoseStamped"
    }
}
EOF
```

Here, we define a goal point which is at 1 meter on the right from the current robot pose (i.e. referred to the `base_link` frame). In particular, the `value` field is formatted like the `PoseStamped` ROS message.

6.5 Demo Execution

At this point, everything is ready for the execution of the demo. The reader can send the `PATCH` command explained before. Looking at the Gazebo window, the robot will start to move to the given point. Naturally, we can also check the odometry value that is continuously updated by the `firos` node. This information can be easily retrieved by querying the Orion Context Broker, as shown in Section 5.4. Nevertheless, the value of the `odometry_filtered` attribute is quite complex compared to the previous example. In fact, it contains several data, such as position, twist, and relative covariance matrix. Hence, it is not so easy to read by humans. For this purpose, our repository contains a utility python program that parses this odometry message and outputs only the data relevant to this example case (i.e. `x` and `y` position). So, launch the program using:

```
$ rosrun fiware_polling_command get_odometry_x_y.py
```

In this way, the reader can easily check the updated odometry values[19]. We have demonstrated a basic Cloud Robotic infrastructure
to control a ROS-based robot using the FIWARE framework. In our example, we used command line tools to control the robot to provide a deep insight of the adopted technologies. Obviously, a real Cloud Robotic application which aims to control a robotic system will have a graphical user interface that can be easily implemented using web technologies (e.g. HTTP-based) or with pre-configured FIWARE instances (see Section 3).

7 Conclusion and Future Works

Nowadays, the importance of the Cloud technologies is attested by the wide adoption of this technology in our everyday life. All the main Hi-Tech companies provide a cloud-based solution in the form of Software-as-a-Service, Platform-as-a-Service, and Infrastructure-as-a-Service. On the other hands, robotics is one of the future emerging markets. Despite this, a clear solution that links these two distinct worlds is not clearly affirmed yet.

The present chapter proposes the use of the FIWARE framework for the development of Cloud Robotics applications using ROS-based robots. In particular, it plainly shows how to integrate this Cloud technology through detailed walkthrough tutorials. The chapter presents the steps to create and use a FIWARE Orion Context Broker running on a VM in FIWARE platform, and its integration with the ROS framework using the `firos` package. The reader has the possibility to follow the walkthrough developing a simple *Hello World* example. Finally, a more concrete Cloud Robotics application to control a simulated robot sending a goal and reading odometry data is presented.

The use of the FIWARE framework allows to easily build modular Cloud application, enhancing the proposed system with additional features, the so-called

[19] A video demonstration of this tutorial is available at `https://youtu.be/czhD-krCRZc`, visited on December 2017.

FIWARE Generic Enablers. Among this, a developer can extend the system with a powerful web-based user interface, employ tools for analyzing context information, and modules designed for addressing security and privacy requirements. In addition, FIWARE can represent a common infrastructure between IoT and robotics.

Furthermore, another aspect that emerges from this chapter is the well-know visibility issue between machines connected on different networks across the Internet. This problem can be addressed in several ways as, for instance, setting a dedicated VPN. However, it can be painful and needs to be configured for every machine acting on the system. The solution detailed in this tutorial proposes the development of a *polling* node, which periodically queries the Cloud resource asking for commands. Clearly, this method can be suitable for most of the IoT devices that typically perform simple actions. More realistically, Cloud Robotics applications need a different and optimized mechanism. A viable option is represented by the modern web-based communication technologies such as WebSockets that implements a low-latency, bidirectional communication layer between clients (web browsers) and servers. Concerning ROS, the Robot Web Tools (RWT) [13] project already offers a ROS interface for the WebSocket transport layer through the `rosbridge_suite` package. A practical use case example of the RWT adoption in a Cloud Robotics teleoperation system can be found at [14].

An extension of the FIWARE framework with the development of a module that integrates the `rosbridge_suite` will concern our future works. It will enhance a FIWARE robotic application with real-time capabilities, allowing to optimize the streaming of a huge amount of data.

Acknowledgement

This work was supported by the ACCRA Project, founded by the European Communitys Horizon 2020 Programme (H2020-SCI-PM14-2016) - grant agreement No. 738251.

Bibliography

[1] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[2] Tully Foote. Community Metrics Report. http://download.ros.org/downloads/metrics/metrics-report-2017-07.pdf. Accessed on December 2017.

[3] Ben Kehoe, Sachin Patil, Pieter Abbeel, and Ken Goldberg. A survey of research on cloud robotics and automation. *IEEE Transactions on automation science and engineering*, 12(2):398–409, 2015.

[4] Guoqiang Hu, Wee Peng Tay, and Yonggang Wen. Cloud robotics: architecture, challenges and applications. *IEEE network*, 26(3), 2012.

[5] Koji Kamei, Shuichi Nishio, Norihiro Hagita, and Miki Sato. Cloud networked robotics. *IEEE Network*, 26(3), 2012.

[6] Rajesh Arumugam, Vikas Reddy Enti, Liu Bingbing, Wu Xiaojun, Krishnamoorthy Baskaran, Foong Foo Kong, A Senthil Kumar, Kang Dee Meng, and Goh Wai Kit. Davinci: A cloud computing framework for service robots. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3084–3089. IEEE, 2010.

[7] G. Mohanarajah, D. Hunziker, R. D'Andrea, and M. Waibel. Rapyuta: A cloud robotics platform. *IEEE Transactions on Automation Science and Engineering*, 12(2):481–493, April 2015.

[8] Markus Waibel, Michael Beetz, Javier Civera, Raffaello d'Andrea, Jos Elfring, Dorian Galvez-Lopez, Kai Häussermann, Rob Janssen, JMM Montiel, Alexander Perzylo, et al. Roboearth. *IEEE Robotics & Automation Magazine*, 18(2):69–82, 2011.

[9] Dominique Hunziker, Mohanarajah Gajamohan, Markus Waibel, and Raffaello D'Andrea. Rapyuta: The roboearth cloud engine. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 438–444. IEEE, 2013.

[10] Sten Hanke, Christopher Mayer, Oliver Hoeftberger, Henriette Boos, Reiner Wichert, Mohammed-R Tazari, Peter Wolf, and Francesco Furfari. universaal–an open and consolidated aal platform. In *Ambient assisted living*, pages 127–140. Springer, 2011.

[11] T. Zahariadis, A. Papadakis, F. Alvarez, J. Gonzalez, F. Lopez, F. Facca, and Y. Al-Hazmi. Fiware lab: Managing resources and services in a cloud federation supporting future internet applications. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 792–799, Dec 2014.

[12] F Herranz, J Jaime, I González, and Á Hernández. Cloud robotics in fiware: A proof of concept. In *International Conference on Hybrid Artificial Intelligence Systems*, pages 580–591. Springer, 2015.

[13] Russell Toris, Julius Kammerl, David V Lu, Jihoon Lee, Odest Chadwicke Jenkins, Sarah Osentoski, Mitchell Wills, and Sonia Chernova. Robot web tools: Efficient messaging for cloud robotics. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 4530–4537. IEEE, 2015.

[14] Alessandro Manzi, Laura Fiorini, Raffaele Limosani, Peter Sincak, Paolo Dario, and Filippo Cavallo. Use case evaluation of a cloud robotics teleoperation system (short paper). In *Cloud Networking (Cloudnet), 2016 5th IEEE International Conference on*, pages 208–211. IEEE, 2016.

Authors Biographies

Raffaele Limosani received the Master Degree in Biomedical Engineering at University of Pisa on July 2011 and received the PhD in Biorobotics (cum laude) from the Scuola Superiore Sant'Anna in November 2015. During his PhD, he was a visiting researcher at the ATR (Advanced Telecommunications Research Institute International) Laboratory, Japan. Currently he is a post-doc at the BioRobotics Institute of the Scuola Superiore Sant'Anna. His research fields are Robotic Navigation, Human Robot Interaction and Mobile Manipulation, especially in unstructured environments.

Alessandro Manzi received the MSc in Computer Science from the University of Pisa, Italy in 2007, and received the PhD in BioRobotics (cum laude) from the Scuola Superiore Sant'Anna in 2017. Currently, he is a post-doc at the BioRobotics Institute of the Scuola Superiore Sant'Anna. His research interests include machine learning, computer vision, data processing from depth cameras, robotic navigation, and perception.

Laura Fiorini received the Master Degree (with honours) in BioMedical Engineering at University of Pisa on April 2012 and the PhD in Biorobotics (cum laude) from the Scuola Superiore Sant'Anna in February 2016. Currently she is a post-doc at the BioRobotics Institute of the Scuola Superiore Sant'Anna. She was a visiting researcher at the Bristol Robotics Laboratory, UK. Her research interests include Ambient Assisted Living, Cloud Service Robotics, ICT system for cognitive activation, pattern recognition, signal processing and experimental protocol.

Paolo Dario received his Dr. Eng. Degree in Mechanical Engineering from the University of Pisa, Italy, in 1977. He is currently a Professor of Biomedical Robotics at Scuola Superiore Sant'Anna in Pisa. He has been Visiting Professor at prestigious universities in Italy and abroad, like Brown University, Ecole Polytechnique Federale de Lausanne, Waseda University, University of Tokyo, College de France, Zhejiang University. He was the founder and the Coordinator of the BioRobotics Institute of Scuola Superiore Sant'Anna, where he supervises

a team of about 120 researchers and Ph.D. students. He is the Director of Polo Sant'Anna Valdera. His main research interests are in the fields of BioRobotics, medical robotics, micro/nanoengineering. He is the coordinator of many national and European projects, the editor of special issues and books on the subject of BioRobotics, and the author of more than 200 scientific papers.

Filippo Cavallo MScEE, Phd in Bioengineering, is Assistant Professor at BioRobotics Institute, Scuola Superiore Sant'Anna (Pisa, Italy), focusing on cloud and social robotics, ambient assisted living, biomedical processing, wireless and wearable sensor systems. He participated in various National and European projects, being project manager of Robot-Era, AALIANCE2 and Parkinson Project. He was visiting researcher at the the EndoCAS Center of Excellence, Pisa; at the Takanishi Lab, Waseda University, Tokyo; at Tecnalia Research Center, Spain. He was granted from the International Symposium of Robotics Research Committee as Fellowship Winner for best PhD thesis in Robotics; from the Regional POR FSE 2007-2013 for a 3-years Research position at The BioRobotics Institute; from the ACCESS-IT 2009 for the Good Practice Label in Alzheimer Project; from the Well-Tech Award for Quality of Life with the Robot-Era Project. He is author of various papers on conferences and ISI journals.