# Robotics Documentation

*Release 1.0*

FIWARE-RoboticsGE

Mar 01, 2018

# Contents

Contents:

CHAPTER 1

# User & Programmer Guide

This guide tries to show the user how to use Robotics. Robotics is composed by two components:

- `RCM` (Robot Clone Manager) the distributed platform which manages robots
- `Firos` the bridge which connects robots to the FIWARE world

This is important to remember because a user of Robotics can be from one of the two worlds, the robot world managed by RCM and the fiware world accessed and linked through firos.

## 1.1 Enable the platform

The first thing to explain is how to enable the platform. A machine become the platform when you install the main rcm platform agent, i.e. the rcm master (look at the *Installation & Administration Guide* to see how to do that). Being the platform means that this machine (or master, as we call it) will be the center of all the user will create in its robotic world: we are speaking of simulated robots for now because we still don't have added physical robot in that world but can be physical robot or machines that works as robots in the platform; machines working as robots only means that you run the rcm robot agent into them: in any case we'll see later how to *add robot to the platform*. At this point the platform is enabled and you can see that calling the web services exposed by the master:

```
~$ curl http://public_ip_master
```

The response is a simple html page remembering that rcm is responding and so is up and running. The web server runs only in the master so you can check only the master installation in this way, but you know that a minimal configuration of the platform is available. You can have a look at what you have in your platform contacting platform_instance and see that your platform doesn't have robot for now:

```
~$ curl http://public_ip_master/platform_instance/read
```

The response is a json formatted response with the list of robots available in the platform so you have an empty list for now.

## 1.2 Add robots

Now that your platform is available you can add the robots: these could be physical robots, physical computers or virtual machines, but the important thing is that they have to run the rcm robot agent. Before that an rcm agent would be recognized by the platform you have to do the provisioning of the robot; you have to tell the platform the `name` of your robot and what will be its role, what it can do: we call this `service logic` (see *Create brains* to better understand the concept and how to create a service logic). You'll provide a new robot calling platform_instance in the rcm master instance:

```
~$ curl -H 'Content-Type: application/json' -d '{"pi_name":"name_r","pi_service_logic
↪":"name_s_l"}' http://public_ip_master/platform_instance/provisioning
```

You can do this before or after running the machine with the rcm robot agent but only when this operation will be done the robot will be available on the platform and can be used. The name of the robot must be unique because this will be the identifier for the robot and all the components generated for that robot; the service logic is mandatory too otherwise the provisioning fails. The response provides the result of the operation but in any case you can see the new platform situation calling platform_instance in the rcm master instance:

```
~$ curl http://public_ip_master/platform_instance/read
```

This time you could see the newly added robot in the no more empty list of available robots. In this case you can see the state of the rcm robot and verify if it's up and running:

- `connected` is true when you switch on your provisioned robot. The agent on the robot has contacted the master to advertise its presence in the platform and the master agent recognise him as one the the robot provisioned by the user
- `paired` is true when the robot is associated with a server (as its brain) and its role is defined (the service logic used for the robot at provisioning time is running)

## 1.3 Create brains

In this section will speak about `service logic`, its meaning and how to create a new service logic. Often the robot is a machine less computationally equipped but that needs more power that he has in order to do all what he has to do. The robot is associated with an additional brain at the boot time (after the installation of the rcm robot agent and the robot provisioning) and and has a role in the platform which means what he can do in the platform: this is defined by the service logic. In terms of robotic world the logic units are tied with the underlying ROS and can be of two types:

- `Node` a process that perform computation and does one of the task of what the user want to let the robot do. Every task can involve more other nodes as sub task: to find more information about ROS nodes see the documentation. We define this underlying component as `service node` and can be accessed in the service logic as one of the two types of service items
- `Launcher` a sort of aggregator, something that provide a common starting point for nodes in the ROS point of view. We define this as `service launcher` and can be accessed in the service logic as one of the two types of service items

The service logic provides a context in which the underlying elements can run and live to give the feature or brain the robot needs. There are many service items already available through the full installation of the underlying layer but are not listed through the platform in the fiware version: you can see them looking in the ROS environment or documentation to see what the installation deploys into the machine.

The service logic is composed by 3 parts:

- `name` the name that identifies the service logic into the platform

- `context` the environment in which the elements of the service logic run and live. We call them `service spaces` and are a sort of containers in which the service items work. This is not configurable but every service logic instance and so every robot create and use one service space identified by the same name used to identify the rcm robot
- `list of service items` the elements that define the behaviour of the robot and in terms of ROS point of view what is launched in the service logic context

### 1.3.1 Manage service logic

There are three operations that can be done on a service logic:

- create a new service logic
- delete an old and no more used service logic
- retrieve the set of available service logics

You can create a new service logic calling service_logic in the rcm master instance:

```
~$ curl -H 'Content-Type: application/json' -d '{"slg_name":"name_s",...}' http://
↪public_ip_master/service_logic/provisioning
```

The important thing to remember in this operation is that you have to provide 2 of the 3 parts of the service logic we listed before at the end of the section *Create brains*. The `slg_name` parameter is mandatory and as we said before identifies the service logic in the platform: when you do the provisioning of the robot you have to provide this name in `pi_service_logic` parameter and the platform starts an instance of that service logic when the robot is turned on. Starting an instance means only that in the context (what we called service space in the previous section) will be launched all the nodes and launchers defined for this service logic. In the provisioning of the service logic you have to provide a complete list of items you need so `sn_list` and `sl_list` should be added in json format. You could have only nodes or only launchers so you can use an empty list for the parameter you don't use.

`sn_list` and `sl_list` stand for service node list and service launcher list. All their elements follow the form of their type of element in the underlying layer so

- `sn_package` and `sl_package` will be the names of the service node (sn) or service launcher (sl) packages, the packages of the ROS nodes or launchers
- `sn_type` will be the type of the service node, the name of the executable or python script representing the type of the node in ROS
- `sl_file_launcher` will be the file name of the service launcher, the name of the scripting file representing the launcher in ROS
- `sn_name` and `sl_name` will be the names assigned to the service node or the service launcher
- `sn_params` will be the parameters passed to the service node or launcher

The only parameters that are specific to rcm platform are:

- `sn_side` and `sl_side`: they represent the side where a node or a launcher will be run. The meaning of this field is tied to the meaning of the context or service space: the service space is a logical container which represents the link between two machines, a server and a robot, and has a sort of manager or main component that in the underlying ROS is called `roscore`. This component will be on server side by default but all the other node and launcher can run on server side or robot side. You have to decide where to launch the elements but remember that the additional brain and the machine more powerful should be the server and should be the preferred side where to launch more resource greedy processes

The information you pass to the platform is not verified so if you put a not existing node into the service logic the result will be that the platform will be unable to correctly start the robot using that service logic. In any case the result of the service logic provisioning will be OK if the syntax of the operation was right so be careful when you create a service logic to provide the available items and correct parameters.

At any moment you can see the service logic that are available in the platform and how is composed what you created looking at service_logic in the rcm master instance:

```
~$ curl http://public_ip_master/service_logic/read
~$ curl http://public_ip_master/service_logic/read?slg_name=name_s_l
```

The first give you an overview of the service logic in the platform (those created by default and those created by you) and the second give a more detailed overview about a specified service logic referred by name.

The last operation you can do is the deletion of the service logic you created if you are not satisfied or you want change something (no changes can be done, so if you want add some nodes or change a launcher you used, you have to remove the service logic and repeat the provisioning with the same name but with the newly designed structure). The deletion can be done calling service_logic in the rcm master instance:

```
~$ curl -H 'Content-Type: application/json' -d '{"slg_name":"name_s"}' http://public_
↪ip_master/service_logic/r_provisioning
```

## 1.4 Give the brain a body

When you finished to design the brain for your robot you have to provide a body to that brain and you do that when you do the provisioning of the robot. All that you set to run in the service logic will be launched where you asked when the robot is switched on after the robot provisioning. You can check if all went well looking at platform_instance in the rcm master instance:

```
~$ curl http://public_ip_master/platform_instance/read
```

The service logic was good and the provisioning went well if the `paired` field becomes true. This change of state require some time so wait before considering the operation a failure. Even in the case of robot as it happens in service logic case, if you want to change something about the robot you have to remove the robot and provide again with the new values. If you change the name you have to change the name in the configuration file of the rcm robot to match the name you newly provide. In order to remove the robot you can call platform_instance in the rcm master instance:

```
~$ curl -H 'Content-Type: application/json' -d '{"pi_name":"name_r"}' http://public_
↪ip_master/platform_instance/r_provisioning
```

## 1.5 Connecting to the Fiware world

In order to understand and provide the connection to the fiware world you have to know that this link is done through firos and you need to put that part in your custom service logic to do it. During the master installation the wizard ask you if you want to enter the fiware world and install the firos package (see *Installation & Administration Guide*). If you require that, firos, rcm_driver and robotics_msgs will be deployed in the ROS workspace used by the rcm platform to run the underlying nodes and launchers. You can see those 3 elements as service nodes needed to exchange information between the robotic world and fiware world. Rcm platform speaks to rcm_driver to tell firos what's happening in the rcm platform and firos communicates those information to the fiware context broker. Rcm_driver speaks to firos in the ROS environment using the language specified in robotics_msgs. All this explanation is intended to let you know that if you want to connect with fiware in your custom made service logic you have to put those 3 nodes in it. Moreover those 3 service nodes are deployed in the master and are available only there, so when you create your service logic you must tell it to run them on the server side. If you do that when you turn on your robots they are notified in fiware world and and an entity of each robot will be automatically available there.

## 1.6 Robots in context broker

**Firos will perform a mapping between ROS and context broker following these rules:**

- Each robot will be an entity, its type will be ROBOT and its id the name of the robot.
- The topics from the robot will be translated into attributes, its type will be the type of the topic and the name will be a slightly modified version of the name of the topic.
- Each topic will have a timestamp named firostimestamp.
- If you want to send a command from context broker to a robot, you must list the name of the attribute to be sent into the value of the attribute COMMAND (type COMMAND).

```
{
    "type": "COMMAND",
    "name": "COMMAND",
    "value": [
        "pose"
    ]
}
```

### 1.6.1 Firos whitelist

Firos selects which topics will be mapped into context broker throught its whitelist, which can be configured in the whitelist.json file. This file has the following format:

```
"name_of_the_robot": {
    "publisher": ["list_of_topics_to_be_received_from_context_broker"],
    "subscriber": ["list_of_topics_to_be_sent_to_context_broker"],
}
```

For example:

```
"turtle\\w+": {
    "publisher": ["cmd_vel"],
    "subscriber": ["pose"]
},
"robot\\w+": {
    "publisher": ["cmd_vel.*teleop", ".*move_base/goal", ".*move_base/cancel"],
    "subscriber": [".*move_base/result"]
}
```

Both the name of the robot and the name of the topics can be regular expresions.

### 1.6.2 Robots descriptions

Robots may have some public files so users can understand some characteristics or even use their devices. All the references contained in this file can be published on the context broker; to do so, just configure the robotdescriptions.json file following this example:

```
"turtle1": {
    "descriptions": [
        "http://wiki.ros.org/ROS/Tutorials/UsingRxconsoleRoslaunch",
        "http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes"
    ]
}
```

This data will be inserted in the entity as follows:

```
{
    "type": "DescriptionData",
    "name": "descriptions",
    "value": "http://wiki.ros.org/ROS/Tutorials/UsingRxconsoleRoslaunch||http://wiki.
↪ros.org/ROS/Tutorials/UnderstandingNodes"
},
```

CHAPTER 2

# Installation & Administration Guide

This guide tries to define the procedure to install and configure Robotics. Robotics is composed by two components:

- `RCM` (Robot Clone Manager) the distributed platform which manages robots
- `Firos` the bridge which connects robots to the FIWARE world

It is important to remember that installing RCM means install an agent on each device involved in the platform. From now on we call the platform 'rcm platform' and the single agent on a device 'rcm instance'. There should be at least one kind of machine involved in the rcm platform, a server who hosts the rcm instance we call 'rcm master'; we call this machine master.

This is the base configuration in which you can use only simulated robots on that machine. In order to add real robots to the rcm platform there should be added other kind of machines called robots. Each robot hosts an rcm instance called rcm robot.

## 2.1 Requirements

RCM and firos must have the following software installed:

- `Ubuntu 14.04 LTS/12.04 LTS` (other versions are not tested)
- `Python 2.7`
- `ROS Indigo/Groovy` (other versions are not tested)

In addition, the master must have `OpenVPN server` and `Sqlite` in order to install rcm master and the robot must have `OpenVPN client` in order to install rcm robot. Firos need `Nodejs 0.10` but is not mandatory and can be configured to not use the additional features that requires it.

Python and sqlite should be already available in the Ubuntu OS but in any case for all the software requirements refer to the products pages to find out how to download, install and configure them. In addition to the instruction provided by the products you have to take care of a couple of things regarding RCM:

- rcm master internally runs a web server to expose the RDAPI. This web server is forced to use the network interface called 'eth0' so make sure that this is the main network interface accessible by the user on the master.

- rcm instances, as agents of the platform, need to deal each other and to do so they use the VPN provided by OpenVPN server installed on the master. They are forced to use the network interface called 'tun0' so make sure to configure properly the OpenVPN server and clients to use this network interface. Moreover remember to emit the certificates for all the OpenVPN clients using the OpenVPN server on the master in order to use the same VPN for all the platform.

## 2.2 Distribution

Starting from the source code you can create the distribution packages needed to install all the components of Robotics; the main tool to do this is the python script `create_robotics_pkg` found in the `scripts` folder. The following are all the options you can have with the tool:

- without options: this is the more completed distribution package because contains the rcm master agent and all the packages needed for the communication in fiware (firos, rcm_driver and robotics_msgs). It creates `robotics.zip` with the root folder called `rcm_platform_master`
- `master`: this is the distribution package with only the rcm master agent. It creates `rcm_platform_master.zip` with the root folder called `rcm_platform_master` but the packages needed for the communication in fiware must be created and installed separately
- `robot`: this is the distribution package with only the rcm robot agent. It creates `rcm_platform_robot.zip` with the root folder called `rcm_platform_robot`
- `firos`: this is the distribution package with only firos. It creates `firos.zip` with the root folder called `firos`
- `rcm_driver`: this is the distribution package with only rcm_driver. It creates `rcm_driver.zip` with the root folder called `rcm`. This is a driver component to let the platform communicate with firos through ROS
- `robotics_msgs`: this is the distribution package with only robotics_msgs. It creates `robotics_msgs.zip` with the root folder called `robotics_msgs`. This is the component providing the intercommunication language for firos and rcm_driver through ROS

```
~/rcm_platform_master$ ./create_robotics_pkg
```

## 2.3 Installation

As we saw in the previous section the distribution packages are provided in compressed files. In this section we will see their structure and how to use them to install Robotics. The robotics.zip and rcm_platform_master.zip are substantially the same thing as internal structure:

```
rcm_platform_master/
        rcm_platform/
                __init__.py
                rcmp_command.py
                rcmp_event.py
                rcmp_ext_connector.py
                rcmp_inter_communication.py
                rcmp_ms.py
                rcmp_node.py
                rcmp_platform_command.py
                rcmp_robotics_data.py
                rcmp_service_command.py
                rcmp_wd.py
        ros/
                rosconsole.config
        install.sh
        sub_inst.sh
        uninstall.sh
        sub_uninst.sh
        rcmpd
        rcmp_n
```

The only difference between the two packages is that the robotics.zip in the root folder has firos.zip, rcm_driver.zip, and robotics_msgs.zip too. These are the distribution packages of the other components used to complete the Robotics and are embedded into the main package.

The rcm_platform_robot.zip shares the same structure of the previous two but loses the two python modules related to the web server and the database that are only available in the rcm master agent:

```
rcm_platform_robot/
        rcm_platform/
                __init__.py
                rcmp_command.py
                rcmp_event.py
                rcmp_inter_communication.py
                rcmp_ms.py
                rcmp_node.py
                rcmp_platform_command.py
                rcmp_service_command.py
                rcmp_wd.py
        ros/
                rosconsole.config
        install.sh
        sub_inst.sh
        uninstall.sh
        sub_uninst.sh
        rcmpd
        rcmp_n
```

The other three packages shares the same structure of typical ROS nodes implemented in python with the root folder named with the package name used in ROS environment and the main python entry point named with what is the type of the node in ROS context. Usually the entry point and all the python modules goes in the scripts folder; configuration files are managed internally by the node so the provider decides where put them. Special folders containing the syntax of messages exchanged by the nodes are in ‘srv’ or ‘msg’ folder, depending on the type of messages that can be exchanged (more detailed and clear explanation can be found on ROS documentation).

Firos:

```
firos/
        config/
        scripts/
                include/
                        ...
                core.py
                ...
        CMakeLists.txt
        License.txt
        package.xml
        README.md
```

rcm_driver:

```
rcm/
        scripts/
                rcm_driver.py
        CMakeLists.txt
        LICENSE.txt
        package.xml
        README.md
```

robotics_msgs:

```
robotics_msgs/
        msg/
                CB_Event.msg
                Robot_Event.msg
        srv/
                FIROS_info.srv
        CMakeLists.txt
        LICENSE.txt
        package.xml
        README.md
```

The rcm agent packages contain the scripts for installing and uninstalling all the packages: this is because during the installation process the scripts create the folder used by RCM to manage the ROS nodes and so all the other components of Robotics. In any case you can install all the other components manually simply extracting the content of the distribution packages into `rcmp_ws/src` folder (this folder is already available only if you have already installed the rcm platform part).

For what concern the installation of robotics.zip, rcm_platform_master.zip or rcm_platform_robot.zip you have to extract the content of the selected component and run the install.sh found under the root folder:

```
~/rcm_platform_master$ ./install.sh
```

The procedure installs the package dependences and places all the needed files in the right places: some of these steps require the administration permission so after the first steps it will be asked you the sudo password. The installer provides a step by step configuration process and can be used as a wizard to perform changes in the RCM configuration files. At the end of the process it will ask if it has to start the 'rcm daemon'. You can also change directly the available configuration files or create them from scratch. If you don't let the installer start the rcm daemon or you have manually changed the configuration files, you have to start or restart the rcm daemon as follow:

```
~$ sudo service rcmpd start
```

or

```
~$ sudo service rcmpd restart
```

### 2.3.1 Package dependences

RCM has two main dependences that are automatically installed using apt-get tool during the installation:

- `python-netifaces` for the access of the network interfaces (tun0, eth0)
- `python-twisted` to implement the web service

python-twisted is used as web server to provide the web services needed to implement the RDAPI available only through the rcm master so this package will be installed and used only in that rcm instance.

### 2.3.2 Installation files

As said before, the installer places all the needed files in the installation package in the right places and creates all the needed folders; in this section we will list all the files and folders arranged by the installation process and will give a brief explanation for each of them.

- `~/rcmp_ws/` this folder is created in the installer's home and is used by the platform as default workspace for ROS custom nodes and launchers. All the packages of the components of Robotics that are not rcm platform agents will be put in this folder under `src` subfolder
- `/usr/local/bin/rcmp_n` is the rcm instance start point, the agent itself, that is started as daemon through rcmpd: it uses the python modules in the python package `rcm_platform` to do its work
- `/usr/local/lib/python2.7/dist-packages/rcm_platform/` is the core python package that implements all the feature of RCM and contains all the python modules. The content will be the following:

  ```
  __init__.py
  rcmp_command.py
  rcmp_event.py
  rcmp_ext_connector.py
  rcmp_inter_communication.py
  rcmp_ms.py
  rcmp_node.py
  rcmp_platform_command.py
  rcmp_robotics_data.py
  rcmp_service_command.py
  rcmp_wd.py
  ```

  In case of rcm robot, rcmp_ext_connector.py and rcmp_robotics_data.py are not available because they are used only by rcm master
- `/opt/rcm-platform/` this folder is created in the opt folder and is the base directory for the agent; it contains the configuration files, the robotics data, the logs and the ROS logging configuration file. Remember that init.cfg is only available in robots while .init.cfg and robotics data are available only in the master
- `/etc/init.d/rcmpd` is the script used to keep the rcm instance as a daemon in the system where is installed; the installer uses update-rc.d tool to insert into the system init structure so that the daemon will be run at start time

## 2.4 Configuration

The configuration process prompts some questions to the user, proposing available responses within brackets in case of selection queries and showing a default response within square brackets (used in case the user presses only Enter key):

```
~/rcm_platform_master$ ./install
The workspace rcmp_ws already exists: do you wanna replace it ([Y]es/no)?
```

As said before we have two kinds of agents in the platform so different questions are proposed in the two cases. Rcm master doesn't have the standard configuration file init.cfg but uses a hidden file .init.cfg that looks the same but with only the list of ports opened for inbound communication; this instance directly controls the robotics data so further creates and uses the sqlite database file named robotics_data.db. Rcm robot uses the standard configuration file and needs a name for the platform instance and the ip address where to find rcm master too.

Don't forget that the list of ports for inbound communication should be configured only if we need ports available from outside the VPN so you have to open the ports on the NAT and firewall to make them available to the platform. Moreover for what concern the name and the ip address configured on the rcm robots is important to remember that the ip address of the master must be the one in the VPN (network interface 'tun0' is used by the instances to communicate each other) and the name of the robot must match the one provided during the provisioning phase (once the rcm master is installed and running exposes RDAPI: to add robots to the platform you have to call the provisioning web service and provide the name there will be used for the robot). For more information about the provisioning process you can see the *User & Programmer Guide*.

### 2.4.1 Platform instance configuration files

In the source repository is available under the folder 'cfg' an example of file configuration called `init_template.cfg`:

```
# This is a template for the configuration file for an RCM platform node.
# Remember that only robots and virtual machines must have this file specified and␣
↪that must be renamed
# with the name init.cfg.

[main]

# the name for the instance (rcm platform node)
# name=instance_name

# specify a list of ports that are previously opened for inbound communication
# inbound_open_ports=10100, 10101, 10102, 10103

# specify the type of the instance (rcm platform node) in case the system we are␣
↪running on is a robot;
# used as flag to distinguish a robot from a vm instance (the master doesn't have␣
↪this configuration file so
# it is already different)
# robot=yes

# the ip of the master rcm platform node
# ip_master=10.xx.yy.1
```

The file is a template of what is created with the wizard procedure and can be used as base for the creation of the needed files from scratch: before every option in the main (and only for now) section there is a brief description of the meaning of that option. The following examples are how the init.cfg file on the robot and the .init.cfg file could look like:

```
# init.cfg

[main]
name=robot1
robot=yes
ip_master=10.11.12.1
```

```
# .init.cfg

[main]
inbound_open_ports=10100, 10101, 10102, 10103
```

### 2.4.2 Used ports

The main port used by all the rcm instance in the platform is 9999 on TCP protocol to communicate each other: it is used inside the VPN channel so there shouldn't be problems about inbound/outbound access and firewall/NAT concerns. Furthermore in case of the master we need the port 1194 on UDP protocol used by OpenVPN server to provide the VPN channel and the port 80 on TCP protocol used by rcm master to provide the interaction with external users. These two ports require inbound and outbound access so you have to open them on your firewall/NAT. Remember that all the ROS nodes in the ROS framework under the RCM platform use a port to communicate with their master but all these communications are inside the VPN channel. During the configuration phase you will be asked for opened inbound ports: these ports are needed only if you need that one ROS node must be accessed from outside the platform. Firos, for example, needs a port opened to be reached by the fiware context broker and communicate with it. Every robot in fiware will live in a logic container called 'service space' so the rcm master managing multiple robots in fiware will need as much opened ports as the number of service spaces: every service space will be associated to a robot and will run a firos instance at master side that will need its own port to let the context broker know its existence. All inbound ports used by ROS nodes in these cases have to be opened on your firewall/NAT: RCM suppose that all the opened inbound ports are properly managed by the network manager of your environment.

## 2.5 Uninstallation

The distribution package of all the rcm platforms provides the tool for uninstalling Robotics. Enter into distribution folder or if no more available take the zip file and extract the content. Run the `uninstall.sh` script to uninstall all the components of Robotics:

```
~/rcm_platform_pkg$ ./uninstall.sh
```

this command removes all the files associated with RCM except the *package dependences*.

## 2.6 Sanity check procedures

In order to see if the installation went well you can check if the rcm master or rcm robot is running:

```
~$ sudo service rcmpd status
/usr/local/bin/rcmp_n is running
```

This is not an exhaustive test because this tells you only that the daemon is up and running, but doesn't know if all the internal steps have started correctly to consider the platform up and running.

### 2.6.1 End to End testing

The main test to verify if the platform is well started is to contact the web server on rcm master. This tells you that the platform is running and the main component can receive the requests from the user:

```
~$ curl http://public_ip_master
<html><body>Welcome to the RCM platform node server!</body></html>
```

Actually the best way to check if all is ok, is to use the read web service always on the web server on rcm master because tells you that the platform is running and rcm master receives the requests. Moreover it gives you an overview of the platform and what is available into it; this means that you can see if the rcm robot you added to the platform are up and running and are connected to the rcm master:

```
~$ curl http://public_ip_master/platform_instance/read
{"reason": [], "result": "OK"}
```

In the example below we have a platform with only the master that answers to the requests and so it's fully operative. Rcm robots are running only if are listed in this response and `connected` is true. At the end they are fully operative only if `paired` is true too. More information about this can be found in the User and Programmer Guide under section *Add robots*.

In order to test if firos is publishing into ContextBroker you can run the following command:

```
~$ rostopic pub -1 s1 std_msgs/String "data: 'test'"  __ns:=end_end_test
```

And then:

```
~$ (curl contextbroker_ip:1026/v1/queryContext -s -S --header 'Content-Type:␣
↪application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <
↪<EOF
{
        "entities": [
                {
                        "type": "ROBOT",
                        "isPattern": "true",
                        "id": "end_end_test"
                }
        ],
        "attributes": [
                "s1"
        ]
}
EOF
```

If everything went right you'll get something like this:

```
{
        "contextResponses": [
                {
                        "contextElement": {
                                "attributes": [
                                        {
                                                "name": "s1",
                                                "type": "std_msgs.msg.String",
                                                "value": "{%27firosstamp%27:␣
↪1443020619.58971, %27data%27: %27test%27}"
                                        }
                                ],
                                "id": "end_end_test",
                                "isPattern": "false",
                                "type": "ROBOT"
                        },
                        "statusCode": {
                                "code": "200",
                                "reasonPhrase": "OK"
                        }
                }
        ]
}
```

Notifications from ContextBroker to firos can be tested by running the following command in one terminal…

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

… and the following command in another terminal:

```
~$ (curl contextbroker_ip:1026/v1/updateContext -s -S --header 'Content-Type:␣
↪application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <
↪<EOF
{
          "contextElements": [
                  {
                          "type": "ROBOT",
                          "isPattern": "false",
                          "id": "end_end_test",
                          "attributes": [
                          {
                                  "name": "p1",
                                  "type": "std_msgs.msg.String",
                                  "value": "{%27data%27: %27`echo $RANDOM`%27}"
                          },
                          {
                                  "name": "COMMAND",
                                  "type": "COMMAND",
                                  "value": ["p1"]
                          }
                          ]
                  }
          ],
          "updateAction": "APPEND"
}
EOF
```

If everything went ok, in the first terminal you'll see something like this:

```
data: random_number
---
```

## 2.6.2 List of Running Processes

This section must be separated into 2 cases because rcm master starts more than one process while rcm robot only one (if you don't consider the ROS nodes and launchers that are processes or bunch of processes too):

```
~$ ps aux | grep rcmp_n
root 1353 0.1 0.3 223740 19876 ? Rl 08:53 0:41 /usr/bin/python /usr/local/bin/rcmp_n
root 2000 0.0 0.2 150016 16968 ? R 08:53 0:06 /usr/bin/python /usr/local/bin/rcmp_n
```

In the example above we have an rcm master with two rcmp_n processes: one is the real server and one is the web server used to expose the RDAPI. In case of rcm robot you'll have only one process, the real server because on those machines you don't have the web server part.

With the following command you can check if the firos process has been brought up:

```
~$ ps aux | grep firos
user    6576  0.1  0.4 513748 18184 pts/10   Sl+  15:38   0:02 python firos_path/
↪scripts/core.py
```

If the firos mapserver has not been configured you'll get the output shown above, otherwise you'll get two process related to firos:

```
~$ ps aux | grep firos
user     6576  0.1  0.4 513748 18184 pts/10   Sl+  15:38   0:02 python firos_path/
↪scripts/core.py
user     6615  0.0  1.8 853320 69308 pts/10   Sl+  15:38   0:00 node firos_path/
↪scripts/mapserver.js 10101 9090
```

Using the following command you can see all the processes launched on the machine relative to ROS and here will have all the service nodes use in the underlying environment (including FIROS).

```
~$ ps aux | grep ros
```

### 2.6.3 Network interfaces Up & Open

As we said in the section about *used ports* Robotics uses a set of ports. You can see the used ports using the following command:

```
~$ netstat -lptu
tcp 0 0 10.82.99.1:9999 *:* LISTEN -
tcp 0 0 192.168.2.74:http *:* LISTEN -
```

In this example we launched the command on rcm master and extracted part of the output: this tells us that the real server communicating with the other rcm agents listens on 9999 port and in a VPN address and the web server runs on 80 (http standard port) and in the address on eth0 network interface (in this case 192.168.2.74).

If you run the same command in an environment where FIROS as been launched you'll find the following lines:

```
tcp 0 0 *:10100 *:* LISTEN -
tcp 0 0 *:10101 *:* LISTEN -
```

The address will be the main address or any-adress (e.g. 0.0.0.0) and the ports will be those in the range you provided during the installation phase.

### 2.6.4 Databases

RCM platform uses a database only on rcm master; this is the only database used in Robotics; it doesn't need much resources and manages a small set of entries so sqlite is the choice for this database. You can find it in the folder named /opt/rcm-platform/ and it is a file called `robotics_data.db`. To access this database you can use the following command:

```
/opt/rcm-platform$ sqlite3 robotics_data.db
```

Remember that this database is in a path that can only be written by root so in the command above you can only read what's inside (sql select query only available in this mode); you can access in writable mode using the same command as before but using sudo: this way is not suggested and the preferable way to change the values in the database is to use the RDAPI. The command sqlite3 is not pre installed so if you want to use you have to install it using the apt-get tool.

## 2.7 Diagnosis procedures

RCM platform provides logging feature to investigate in case of problems; you can find the log files in the folder named /opt/rcm-platform/log. The logger used provides information about what happening in the platform and when the max dimension of the file called rcmp.log is reached it rolls over appending a number after the old version of the file used so you can see what happened before. Only 5 files are written before overwrite the first. To more advanced log features relative to the service nodes (included firos) you can see the ROS documentation about the log.

### 2.7.1 Resource availability

Robotics itself doesn't require much resources, but the problem is that being a platform is supposed to run other things and not only itself. This means that if you run the platform alone you can run it under a small CPU and not much RAM instead if you intend to use with much service nodes running in it you need a really powerful machine. The main idea is to use big machines for the master and small machines for the robots: the robots will do small things and delegate to the master the more resource greedy tasks.

### 2.7.2 Remote Service Access

The only available accesses to Robotics are the exposed APIs:

- RDAPI
- Firos APIs

### 2.7.3 Resource consumption

The resource consumption depends on how you designed your service logic (see the User and Programmer Guide under the section *Create brains* to learn how to do that). If running `top` tool you can see that rcmp_n is using 50% of CPU certainly there is a problem; usually 5% is much when it isn't starting robots, but launching a service logic during the robot startup can increase that value up to 100% of the CPU if the service nodes you added in your service logic need that much. We can say that if you reach some peaks when you're starting a robot is normal, but this is not if that peak is constant. In any case when you look at `top` tool the use of resources is more clear and you can see that rcmp_n increase at start time of the sub processes (when it launches the service nodes and launchers) and then will be the created processes to use the resource if they need them no rcmp_n anymore.

The resource consumption of firos will be minimal, even under heavy messages traffic.

### 2.7.4 I/O flows

The only I/O flows that can concern a user of Robotics should be the user access to the web services and the communication between firos and the context broker. All the other I/O flows are under VPN and should not be a problem.

# CHAPTER 3

# Indices and tables

- genindex
- modindex
- search