Robocologie User Manual

Table of contents:

I. Network:
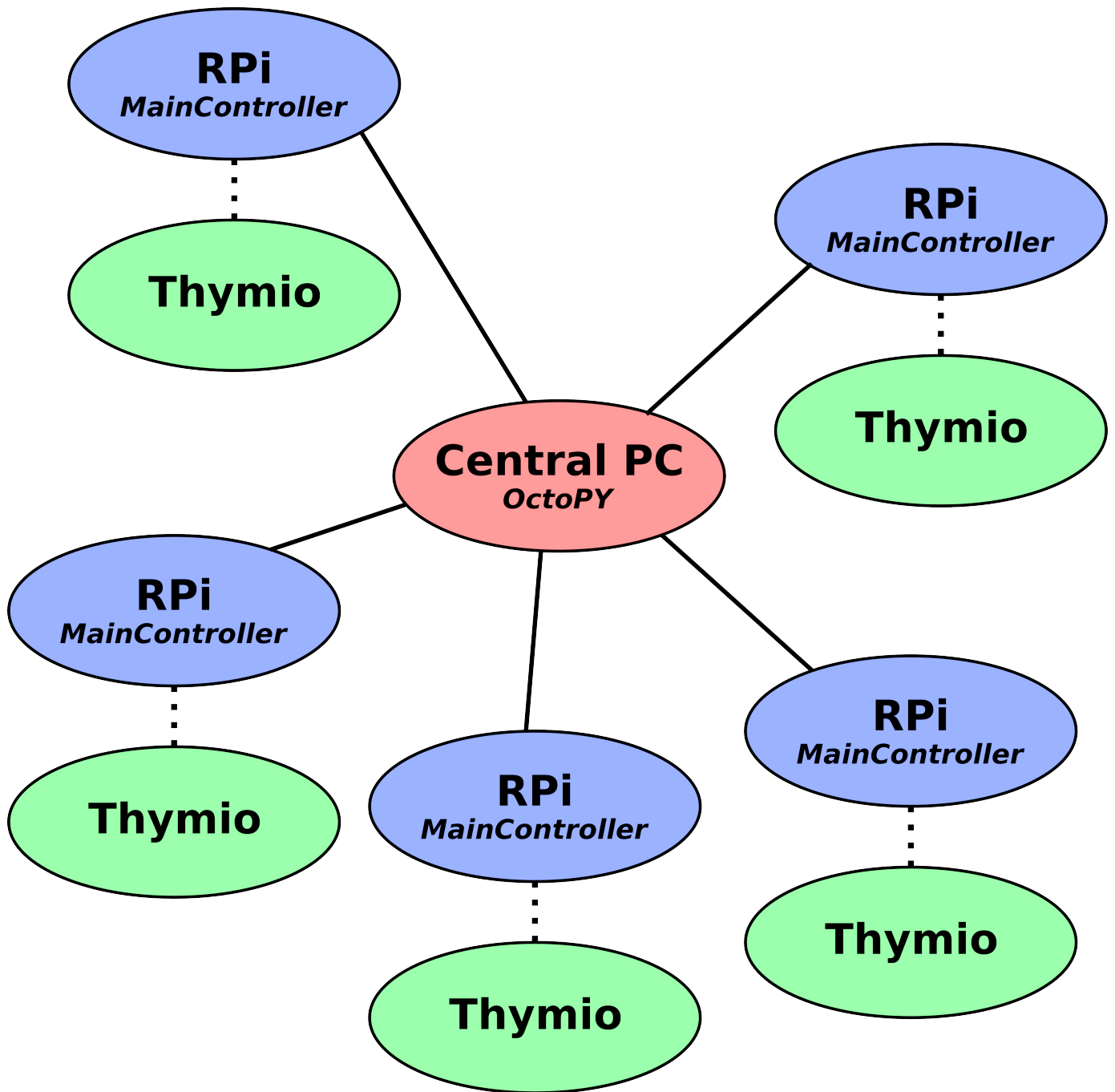
Central PC:

The central PC is used to communicate with every raspberry pi by using the OctoPY software. Communication is achieved with WiFi by a private network that is managed by the central PC.

As such, is it necessary at first to launch this network so that raspberries pi can connect. This is done by having the central PC connect to the access point. The file **/etc/network/interfaces** is configured so that the PC can easily be connected to the access point. To connect to the access point, type the following commands:

```
~$ sudo ifdown eth1 // Not necessary but might be needed when the connection has not
                       previously been properly cut
~$ sudo ifup eth1 // This may display errors but they are not blocking
~$ ping 192.168.0.100 // This is used to confirm that the PC can communicate with the access
                         point
~$ ifconfig // It is also safer to validate that the adresse of the PC is correct: 192.168.0.210
```

When the PC is correctly connected to the access point, it is necessary to start the DHCP server so that raspberries can automatically be issued IP addresses:

```
~$ sudo /etc/init.d/isc-dhcp-server stop // Again may not be necessary but is more cautious
~$ sudo /etc/init.d/isc-dhcp-server start
```

Raspberry Pi:

Raspberries are configured so that they can automatically connect to the network at launch. IP addresses are issued between 192.168.0.110 and 192.168.0.150. If an error occurs, it is possible to manually connect to the network by using **wicd-curses**, which is already installed on the raspberries:

```
~$ wicd-curses
```

Here are the network information:

- **SSID:** NETGEAR_11ng

- **Protocol:** WPA

- **Passphrase:** rpiaccesspoint

II. OctoPY

The software used to remotely control the robots is called OctoPY and can be found at the root of the OctoPY directory. OctoPY is mainly used to send basic instructions (start, stop etc...) to the raspberries. The software can be found at *./thymioPYPI/OctoPY/OctoPY.py*.

Commands:

Look:

Usage: look [range] [-s save_table]

This command is used to look for the raspberries connected on the network. The default range of IPs on which to look is 192.168.0.111-150 but another range can be given through the *range* argument. If the argument *-s* is given, when a raspberry is found, its hostname is retrieved. The association between IP addresses and hostnames is then saved in the hostnames table. This table is saved in a file named *hostnames_table.json* in the same directory. When OctoPY is started, if this file exists, the hostnames table is loaded so that no **look** command is necessary if they was no changes in the raspberries connected to the network.

Send:

Usage: send <message> [hosts_list] [data]

This is the main command that you will use during your experiments. This command is used to send instructions to particular robots (by specifying the *hosts_list* argument) or every known robot. Additionally, the argument *data* can be used to send any necessary data with the instruction.

Messages:

Instructions are sent by specifying a particular message. This message is specified known by its numerical ID, the list of which can also be displayed in OctoPY by taping the TAB key of the keyboard after *send* has been written. Here is the list of the current messages:

- **0 : INIT:** This instruction must always be sent to the raspberries before other instructions can be specified. This is used to launch the main controller on the robots as well as *aseba*. Any other instruction save **OFF** will fail on a non-initialized raspberry.

- **1 : START:** Launches the experiment specified by the configuration filed loaded on the raspberry. If no configuration file has be previously loaded with the **LOAD** message then a default experiment will be launched (a simple braitenberg).

- **2 : PAUSE:** Pauses a started experiment on the raspberry.

- **3 : RESTAT:** Restarts a paused experiment.

- **4 : STOP:** Stops the current started experiment.

- **5 : KILL:** Stops the controller on the raspberry. **INIT** must be sent again before any other messages can be received by the raspberry (save **OFF**).

- **6 : OFF:** Shutdowns the raspberry.

- **7 : SET:** Sets an experiment on the raspberry. The first argument must be configuration file (.cfg file) of the experiment to set then the address(es) of the raspberry(ies).

- **8 : DATA:** Used to send any data to a raspberry.

Hosts list:

The list of raspberries to which the message must be sent can either be precised as an IP address or as a hostname if this hostname exists in the hostnames table (see **look** command). TAB can be used to display the list of possible hostnames.

Query:

Usage: query [hosts_list]

Queries the state of the raspberry. This state can be varied:

- **Down:** The raspberry is off or not connected to the network.

- **Sleeping:** The raspberry is on and connected to the network but not initialized by the **INIT** command.

- **Listening:** The raspberry is initialized.

- **Started:** An experiment is started on the raspberry.

State:

Usage: state [hosts_list]

Gives the state (as defined previously) of a list of raspberries according to the a previously executed **query** command. If no **query** has already been executed then a **query** is done.

Launch:

Usage: launch controller_path [-d detached]

Launches a Controller according to its configuration file (.cfg file). When the controller is launched, no other commands can be used. If the argument *-d* is given, the controller is detached which means it will run in the background (allowing multiple controllers to run at the same time) while OctoPY can still be interacted with. To have more information on how to use Controllers, please see the dedicated section.

Simple Scenario:

This last section is used to illustrate a simple succession of basic instruction in OctoPY, in this case to launch the experiment *SimulationBraitenberg*.

```
>> look -s // Gathers the addresses and hostnames of raspberries connected on the network
>> send 0 // Initialize all the raspberries
>> set braitenberg.cfg // Loads the experiment on every raspberry
>> send 1 pi2no02 // Starts the experiment on the raspberry pi2no02
>> send 1 // Starts every raspberry
>> send 4 // Stops every raspberry
>> send 5 pi2no02 // Kills the controller on raspberry pi2no02
>> send 6 pi2no02 // Shuts the raspberry pi2no02 off
```

Monitoring the raspberries:

While it is not necessary to directly connect on the raspberries, it is often useful to check on the execution of the program. First connect to the raspberry by ssh:

```
~$ ssh pi@<IP address>
```

The username is thus *pi* and the password *pi* also. Then go the folder where the log files are written:

```
~$ cd ~/dev/thymioPYPI/OctoPY/rpifiles/log
```

Then you can read the log file as it is written by typing the following command:

```
~$ tail -F ./MainController.log
```

III. Experiments:

Experiments are the programs launched on the raspberries by the OctoPY and constitute the specific behaviours desired for the robots. OctoPY comes with a framework that is intended to facilitate the integration of new experiments. As such, a minimal implementation of an experiment corresponds to coding the behaviour of the robot at each step of the experiment. As with all of the files intented to be used on the raspberries, the code necessary for experiments is in the *OctoPY/rpifiles* folder. On the raspberries, the corresponding code can be found at *~/dev/thymioPYPI/OctoPY/rpifiles*. Several experiments already exist to serve as easy demonstrations on how to code an experiment. It is advised to create the experiments on the Central PC and then copy them (with the *scp* command) on all the raspberries.

Creating an experiment:

A script exists to quickly create the files necessary to code an experiment: *CreateExperiment.py.*

Usage: python CreateExperiment.py experiment_name

This script creates the basic files and folders that will be used for the experiment *experiment_name*. More precisely, the following files are created:

- **./config_*experiment_name.cfg*:** This file is used to specify the path to the folder and main source file of the experiment which should not be changed. But this file can also be used to specify any parameter that we desire to load with the experiment. Each parameter must be written as follows :

    type parameter_name = parameter_value

    Three different *types* are recognized when parsing this file: *int, float* or *str*. If no *type* is given, then *str* is assumed. Any parameter given in the configuration file may then be accessed in an experiment by using *Params.params.parameter_name*.

- **./experiment_name folder:** This folder contains all the other files created for the experiment. This folder acts as a Python module which is then used by the main controller of the experiments.

- **./experiment_name/__init__.py:** This empty file should often not be changed as it is only used so that the folder is recognized as a Python module.

- **./experiment_name/readme.txt:** This is self explanatory. This file should be used to clearly describe what is the purpose of the experiment.

- **./experiment_name/SimulationExperiment_name.py:** This is the main file where the code will be written. The next section will describe more in the details the content of this

file.

Please note that there exists another script that clone an existing experiment: *cloneExperiment.py*.

Usage: python cloneExperiment.py source_configuration_file experiment_name

The simulation file:

As previously stated, this file is where the major part of the code will be written (in the first time). When created, its content is at follows:

```
import Simulation
import Params


class experiment_name (Simulation.Simulation) :
    def __init__(self, controller, mainLogger) :
        Simulation.Simulation.__init__(self, controller, mainLogger)


    def preActions(self) :
        pass


    def postActions(self) :
        self.tController.writeMotorsSpeedRequest([0, 0])


    def step(self) :
        pass
```

In this file is defined the main experiment class which extends for the Simulation class defined in the framework. It overrides three functions its parent class: *step*, *preActions* and *postActions*.

step is the most important function to write. When an experiment is started, *step* is the function called at each step of the experiment. In consequence, this is where most of the code will be called. *preActions* is used to define all the instructions that must take place when the experiment is started but before the first call of the *step* function. Finally *postActions* is used to code the instructions taking place before the experiment is stopped.

Sending instructions to the Thymio:

Sending instructions to the Thymio is easy as an interface (in the file *ThymioController.py*) is used to transfer the most basic instructions to the Thymio through *aseba*. The *self.tController* property is used to access this controller and interact with the Thymio. These are the instructions that currently exists in the controller:

- **readSensorsRequest:** Reads the values of the proximity sensors. These values must then be accessed thanks to the *getPSValues()* function of *tController*.

- **readGroundSensorsRequest:** Reads the values of the ground sensors. Accessed thanks to the *getGroundSensorsValues()* function.

- **readMotorsSpeedRequest:** Reads the speed values of both motors. Accessed thanks to the *getMotorSpeed()* function.

- **writeMotorsSpeedRequest:** Sends the instruction to move to the robot. The desired speed of each motor is specified as an argument under the form of an array: *[left_motor, right_motor]*.

- **writeColorRequest:** Sends the instruction to change the color of the robot's LED. The desired color is specified as an argument under the form of an array: *[R, G, B]*.

- **writeSoundRequest:** Sends the instruction to emit a sound. The desired sound is specified as an argument under the form of an array: *[frequency, duration]*.

After an instruction has been sent to the controller (in particular after a *read* instruction) it is often necessary to use the *waitForControllerResponse()* function to ensure that the controller has processed the instruction.

Communicating with other robots:

Robots can communicate with each other by using the *sendMessage* from the parent *Simulation* class.

Usage: self.sendMessage(recipients = recipientsList, value = value)

The *recipientsList* corresponds either to a list of IP addresses or hostnames as when using OctoPY.

To receive a message, the function *receiveComMessage* must be overrided.

Usage: def receiveComMessage(self, data)

data["senderHostname"] contains the hostame of the message's sender. *data["value"]* contains the value of the message.

IV. Controllers:

Controllers are used to automate the instructions that are sent by OctoPY so that it not necessary to manually write them. As for experiments, controllers can be created with a script called *CreateController.py* (*./OctoPY/CreateController.py*). And again as for experiments, this script will create a configuration file (.cfg file) and folder containing a file with basic code for a controller:

```
import Controller
import OctoPY
import Params


class ControllerName(Controller.Controller) :
        def __init__(self, controller, mainLogger) :
                Controller.Controller.__init__(self, controller, mainLogger)


        def preActions(self) :
                pass


        def postActions(self) :
                pass


        def step(self) :
                pass


        def notify(self, **params) :
                pass
```

Similarly to experiments, there are three main functions to override: *preActions, step,* and *postActions*. By using the property *self.OctoPYInstance*, it is possible to use OctoPY commands in the same way as with the interactive version. For example, *self.OctoPYInstance.sendMessage* can be used to send messages to raspberries. Typically, *preActions* will be used to initialize and start experiments, *postActions* to stop the experiments and shut the raspberries down and *step* to code the behaviour of the controller while experiments are running.

The most useful behaviour of a controller while experiments are running is to listen to the raspberries. This can be done by *registering* the controller to particular raspberries. This is done

with the function *self.register*. Then, overriding the function *notify* allows to code the behaviour of the controller when notifications are received from the raspberries listened to. You can check the controller **TestNotifications** to have an example on how to code a controller.

As previously explained, a controller can be started by using the **launch** command in OctoPY.

V. Setting a Raspberry:

Most of the raspberries are already configured. But it may be necessary to set new rasperries when the time comes. Fortunately, this is easy.

The first step is to write the default system image on the SD card. The newest images is present on the computer at: *~/rpiImage_pi3noXX_2016_12_15.img*

To write this image on the SD card, first find the devices mounted on the computer:

```
~$ df -h
```

Then, find the partition on which the SD card is mounted. It should be listed as something as */dev/sdc1* and */dev/sdc2*. You can unmount the device by typing the following command:

```
~$ umount /dev/sdc1 // If /dev/sdc1 is the partition on which the SD is mounted
```

You can then copy the system image on the SD by using the following command:

```
~$ dcfldd bs=4M if=<system_image> of=<partition>
```

This should take some time. If you need more information, please read https://www.raspberrypi.org/documentation/installation/installing-images/linux.md

As this image is copied to every raspberry, there are some changes that necessary to do so that each raspberry can be unique. First it is necessary to change the hostname of the raspberry. Each raspberry hostname should read as follows: **pi<raspberry_version>no<raspberry_number>**. For example, for the third raspberry pi 2, its hostname is **pi2no03**. Whatever the raspberry's version (2 or 3), please keep the raspberry numbers successive. To change the hostname of a raspberry, first modify the file */etc/hosts*. Find the line that reads:

```
127.0.1.1          raspberrypi
```

and change *raspberrypi* by the new hostname. Then open the file */etc/hostname* and replace the hostname by this new hostname.

You also need to change the machine-id so that each agent as an unique one. To do that, type the following command:

```
~$ dbus-uuidgen > /var/lib/dbus/machine-id
```

The system image may not have the last version of the code available on the git repository.

As such, you need to pull the last version of the code from the git. There is a basic git account created to pull (and not push) code from the git. Its credentials are:

- **Username:** ThymioPYPI

- **Password:** amac2016

Then, to pull the code move into the */dev/thymioPYPI* directory and type the following command:

```
$~ git pull origin master
```

VI. Files Location:

This section is used to summarize the location of the different files and folders on *thymioPYPI* folder:

- *./robocologieimage*: source code for the tracking and monitoring software.

- *./RFID*: simple test code to use the bar scanners.

- *./TagRecognition:* source code and data for tag recognition. Most of the useful code is also present in *./OctoPY/rpifiles/TagRecognition*.

- *./OctoPY*: folder containing all the files that are used by **OctoPY**.

- *./OctoPY/rpifiles*: folder containing all the files that are supposed to be used on the raspberries.

- *./OctoPY/rpifiles/Files Networking*: folder containing a copy of all the files that are used to configure the network.