RoboShop Project
Delivrable 1
PHAROS: Concepts and Architecture

Noury Bouraqadi, Luc Fabresse, Jannik Laval, and Santiago Bragagnolo

firstname.lastname@mines-douai.fr
http://car.mines-douai.fr

Technical Report 20130415

Dépt. Informatique et Automatique (DIA)
Ecole des Mines de Douai
941 rue Charles Bourseul - C.S. 10838
59508 Douai Cedex
France

This work was done as part of the RoboShop project (2012-2013). It was supported by:

Contents

Chapter 1

Introduction: Robots in Shopping Malls

1.1 Mobile Autonomous Robots

The ROBOSHOP project aims at experimenting the use *mobile autonomous* robots [Par08] in the context of application related to retail. The *mobile* adjective denotes that our robots will act in an open partially known environment which is continuously changing. It is virtually impossible to predict all situations which will confront robots, and even less their sequences because of the combinatory explosion.

Robots that we will use in the ROBOSHOP must also be *autonomous*. Their behavior will be solely driven by the onboard embedded software. A such robot makes decisions on its own without any intervention of a human operator. It should achieve its mission in a way that ensures safety of humans that it might meet, and without damaging neither itself, nor other objects in its environment.

1.2 Robots for Retail

Building robots that are both mobile and autonomous is among the current challenges faced by roboticists. Nevertheless, the current state of the art allows building robotic products for the mass market as exemplified by the robotic vacuum cleaner. Other applications for mobile autonomous robotics exist and some are even turned into actual products such as lawn mowers or swimming pool cleaners.

In the ROBOSHOP project, we aim at showing that robots can be useful in shopping malls. They can attract people by providing them a new breed of services. To illustrate this idea, we will ultimately implement an example of such a service: a robot guide. The robot should be able to guide people within the mall, and provide them with information about available shops and products.

1.3 This Report

This report is the first delivery of the ROBOSHOP project. We describe in chapter 3 (page 7) the software architecture of PHAROS, the ROS client that we have implemented on top of the Pharo programming language. Indeed, we have chosen to comply with ROS (Robot Operating System) which is the *de facto* standard middleware in the robotic community. All the background notions related to ROS required to understand PHAROS are introduced in chapter 2 (page 5).

Chapter 2

ROS

ROS (http://ros.org) is a middleware dedicated to robotics. It enables the interaction between different software possibly written by different developers and running in different processes.

2.1 ROS node

A ROS system is a set of communicating system processes. Each software running in a system process is called a *node*. Nodes are almost never standalone. They can communicate with each other either synchronously (RPC) or asynchronously (publish and subscribe).

2.2 ROS master

A ROS based system relies on a special node called *master*. The ROS master is the entry point for all other nodes, since it allows them discover each other. Because ROS master is playing a so central role, it must be launched before all other nodes. This is done by executing the roscore command line.

The ROS master acts as a naming registry. It stores topics and services registration information for ROS nodes. Nodes communicate with the master to report their registration information. As these nodes communicate with the master, they can receive information about other registered nodes and make connections as appropriate. The master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run.

Nevertheless, nodes "talk" to each other directly. Indeed, the master only provides lookup information, much like a DNS server.

2.3 ROS Messages

Nodes communicate with each other by passing *messages*. 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 (much like C structs)

2.4 Topics and Asynchronous Communications

Asynchronous communications in ROS follow the publish-subscribe model. 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.

2.5 Services and Synchronous Communications

Nodes can communicate synchronously *à la* RPC via *services*. Each service is defined by a pair of messages: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.

Chapter 3

PHAROS

PHAROS is a ROS client developed in the Pharo programming language [NDP10]. Being a client means that it allows building new ROS nodes that communicate with the ROS master as well as with other nodes.

3.1 PHAROS Setup

3.1.1 Building PHAROS

PHAROS relies on several other open source projects programmed in Pharo. The main ones are:

- XMLRPC which provides support for doing Remote Procdedure Call through XML
- OSProcess which provides support to run system commands (such as `rosrun`) from Pharo

To install PHAROS, you should:

1. Download Pharo 1.4 (the virtual machine, the image and the source file) at http://www.pharo-project.org/pharo-download/release-1-4
2. Load the PHAROS code and its dependencies using:

```
Gofer it
  url: 'http://car.mines-douai.fr/squeaksource/PhaROS';
  package: 'ConfigurationOfPhaROS';
  load.
((Smalltalk at: #ConfigurationOfPhaROS) project version: #pharosRoboshop1)
    load.
```

3.1.2 Installing PHAROS on a ROS-ready Computer

At this stage, you need to copy PHAROS to a computer with ROS installed. That is copying the corresponding Pharo files (.image, .change and other files).

Currently, PHAROS supports ROS fuerte[1]. To make the installation simpler, you can download a VirtualBox[2] ready to use virtual machine[3]. All the examples presented in this chapter have been tested in this environment.

Warning: Even if ROS master is located on another machine, you need anyway to install ROS on the machine where PHAROS is run. In fact, the current version of PHAROS uses the rosmsg utility to communicate with other nodes.

3.1.3 Launching PHAROS

We can now launch PHAROS and get a window like the one of Figure 3.1, using the following command in a terminal. We assume that we are logged in as *viki* user on a machine named *ROS*, hence the prompt.

```
viki@ROS$ /home/viki/pharo-vm/pharo /home/viki/pharos.image
```

Warning: Do not use a double-click on the PHAROS.sh file, because ROS environment variables will not be set since the .bashrc is ignored. This is because the current version of PHAROS needs ROS to be installed on the same machine[4].

[1] http://www.ros.org/wiki/fuerte/Installation
[2] https://www.virtualbox.org
[3] http://nootrix.com/2012/09/virtualizing-ros/
[4] PHAROS uses the rosmsg utility to communicate with other nodes

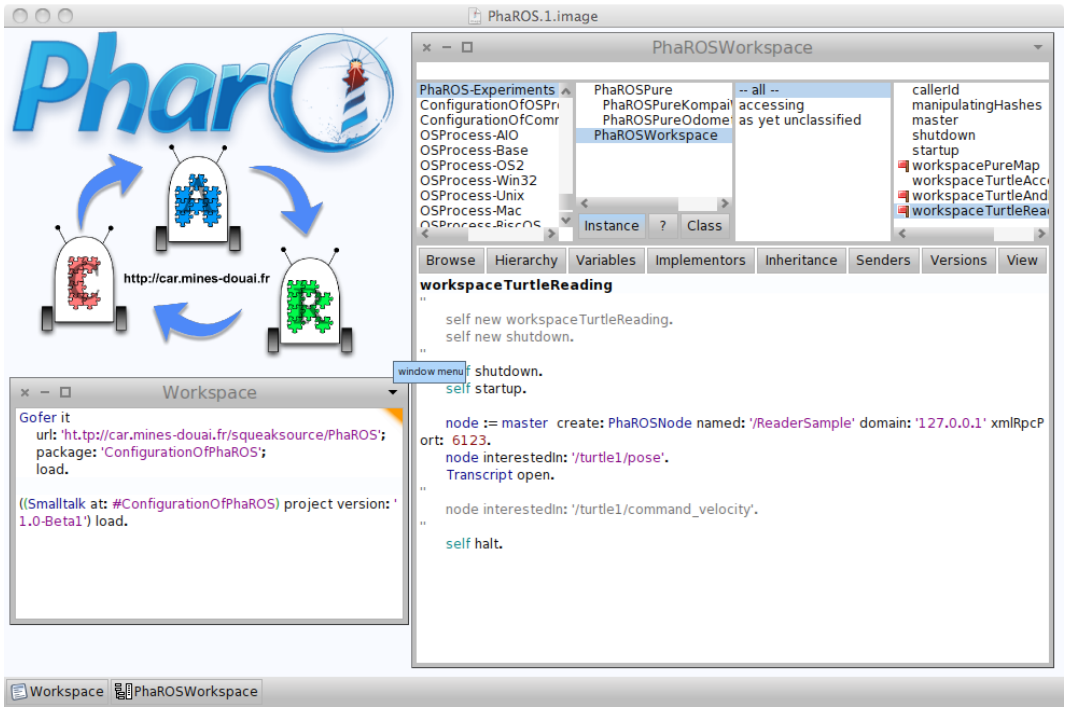

Figure 3.1: Pharo environment with PHAROS code

3.2 General Architecture

Figure 3.2 depicts the general architecture of PHAROS. PHAROS is built around three concepts explained in Chapter 2: ROS Node, ROS master, and ROS Topic. In the PHAROS architecture, they are represented by the three classes: PhaROSMaster, PhaROSNode, and PhaROSTopic. We describe each of them in the following, together with other core classes.

PhaROSMaster. This class is the interface to the ROS master. It allows to discover other nodes inside and outside Pharo.

PhaROSNode. Is a node developed in PhaROS and running inside the current Pharo image. A PhaROSNode contains a publisher and/or a subscriber. A publisher (PhaROSPublisher) allows the node to provide information to other nodes. It uses an output channel (PhaROSOutputChannel) to publish the information. A subscriber (PhaROSSubscriber) gives to the node the information provided by other external or internal nodes. The subscriber is configured by giving a topic, which return the right channel to listen.

PhaROSExternalNode. The class is an interface to any ROS node, external to the current Pharo image. It can be a node which running in another Pharo image or a node developed in any other programming language. This interface allows to receive information from this node or to send information. As for PhaROSNode, it uses the Input or Output Channels to communicate with PHAROS.

PhaROSTopic. A topic represents the dispatcher of channels. When a node asks for information or publishes information, it asks the right instance of PhaROSTopic to give it the correct channel. The topic returns to the node a channel that it can use as output or input.

In case of communication between two nodes inside Pharo, the communication does not go outside Pharo. The mechanism is handled inside Pharo. This provides better performance than if it would use the TCP protocol.

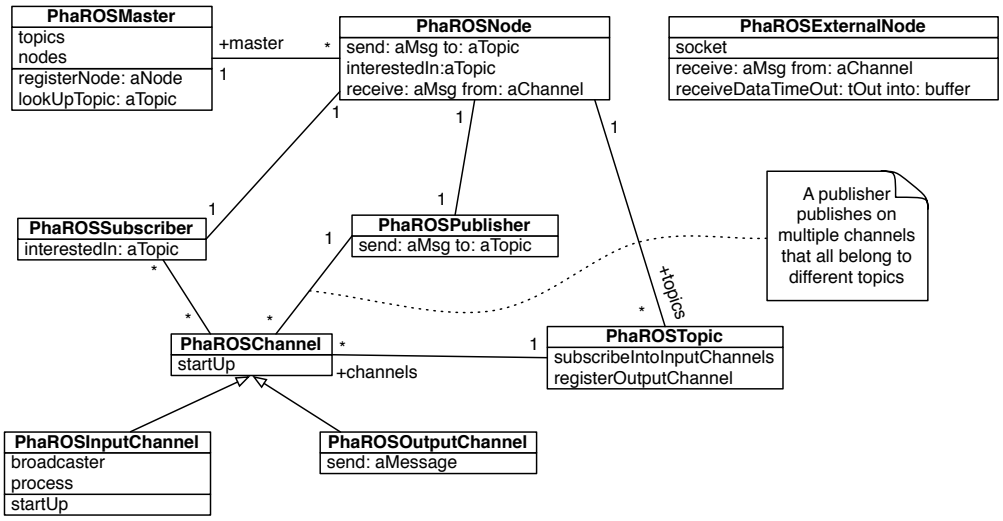

Figure 3.2: Class diagram of PHAROS

3.3 PHAROS in action

In this section we provide examples showing how to use PHAROS for writing nodes. We show step by step what to do on the PHAROS side and command lines to perform on the ROS part.

3.3.1 Preparing ROS

In our examples, we interact with the turtle simulator of ROS. This is actually a node that is provided as part of ROS distribution. Both the turtle simulator and our nodes (i.e. PHAROS) will be running on the same machine.

Before running our examples, we need first to start a ROS master, and a turtle simulator node. To launch the ROS master, evaluate the following command line in a terminal.

```
viki@ROS$ roscore
```

To start the simulator, evaluate the following command line in a terminal.

```
viki@ROS$ rosrun turtlesim turtlesim_node
```

3.3.2 Topic Publisher Node

In this example, we develop a node that publishes velocity messages consumed by the turtle, through the '/turtle1/command_velocity' topic. The full code of the class of our node PhaROSTurtleDriver is provided by the listing 3.1.

```
PhaROSNode subclass: #PhaROSTurtleDriver
    instanceVariableNames: 'process'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PhaROS-Experiments-RoboShopDeliverable1'

PhaROSTurtleDriver>>start
  super start.
  process := [self publishingLoop] newProcess.
  process
    name: 'Turtle Driver';
    priority: Processor userBackgroundPriority;
    resume

PhaROSTurtleDriver>>publishingLoop
|delay|
delay := 1 second asDelay.
```

```
18  [
      delay wait.
20    self
        sendTo: '/turtle1/command_velocity'
22      a: [ :message |
              message angular: 1.0.
24            message linear: 2.0]
    ] repeat
26
PhaROSTurtleDriver>>stop
28  super stop.
    process terminate
```

Listing 3.1: A Topic Publisher Node Definition

The PhaROSTurtleDriver class is defined as a subclass of PhaROSNode. It redefines the start method to fork a process that loops infintely. The loop is defined in method publishingLoop. Every second, it sends a velocity command to the turtle through the '/turtle1/command_velocity' topic.

Publishing into a topic is done using the sendTo:a: message. The first parameter is the name of the topic. The second parameter is a block which argument is the ROS message instance of PhaROSPacket. A ROS message is a data structure with named fields, similar to a struct of the C language. In our example, messages to the '/turtle1/command_velocity' topic correspond to a speed. They have two fields: angular speed and linear speed that are set lines 23-24.

The PhaROSTurtleDriver class overrides the stop method. This is because we need to terminate the process (created by start method) which loops indefinitely. This termination is performed by sending the terminate message to the process (line 29).

To use our class and create an actual ROS node, we need to evaluate the code of listing 3.2 in a workspace. First we create a proxy to the ROS master (line 1). We provide an url with the address and the port of the ROS master. In our example, the ROS master is running on the same machine as our node, so we use the loopback address (127.0.0.1).

```
1  master := PhaROSMaster url: 'http://127.0.0.1:11311/'.
2  node := master
       create: PhaROSTurtleDriver
4      named: '/myTurtleDriver'
       domain: '127.0.0.1'
6      tcpPort: 20202
       xmlRpcPort: 21212
```

Listing 3.2: Running a Topic Publisher Node

Node creation is performed by the ROS master proxy (lines 2 7). We provide it with the node's class (line 2), and the public unique name of the node (line 3). The third

argument is the name or the address of the machine hosting PHAROS. That is the local host in our example (127.0.0.1). Each node communicates with the ROS master and others nodes through TCP and XML RPC[5]. The last two arguments refer to ports for incoming communications to our node for these two protocols.

The node is automatically started. So, the call of method start is implicit. However, to stop any PHAROS node, we need to explicitly evaluate node stop.

Warning: In the current version of PHAROS, a stopped node cannot be restarted. You'd rather need to create another instance of the same class.

3.3.3 Topic Subscriber Node

In this example, we develop a node that logs velocity messages from the '/turtle1/command_velocity' topic. The full code of the class of our node PhaROSTurtleLogger is provided by the listing 3.3.

```
1  PhaROSNode subclass: #PhaROSTurtleLogger
2    instanceVariableNames: ''
     classVariableNames: ''
4    poolDictionaries: ''
     category: 'PhaROS-Experiments-RoboShopDeliverable1'
6
   PhaROSTurtleLogger>>start
8    super start.
     self receiverDelegate: [ :message :channel |
10             Transcript show: message value asString].
     self
12     interestedIn: '/turtle1/command_velocity'
       typedAs: 'turtlesim/Velocity'
```

Listing 3.3: A Topic Subscriber Node Definition

The class PhaROSTurtleLogger includes a single method. It defines a block that is performed each time a ROS message is received (lines 9-10). The block takes two arguments. The first is the ROS message (instance of class PhaROSPacket), while the second is the input channel (instance of class PhaROSInPutChannel) through which the ROS message was received. In this example, we use only the ROS message that we display on Transcript.

To make the node actually receive messages and thus perform logs, we need to register it to some topic. This is done in the last part of the method (lines 11-13) subscribes the node to the '/turtle1/command_velocity' topic.

[5]Communications include registering and handshakes, as well as connection/disconnections

Now we are ready to test our node. We will still be using the turtlesim_node to display what is going on. In addition, we need to run a node that publishes on the same topic. We use the draw_square node from the turtlesim ROS package as following:

```
viki@ROS$ rosrun turtlesim draw_square
```

On the PHAROS side, we need to launch a node instance of our PhaROSTurtleLogger. This is provided in listing 3.4. We don't explain it, since it is nearly identical to the code for the launching the turtle driver (see listing 3.2 page 13).

```
1 master := PhaROSMaster url: 'http://127.0.0.1:11311/'.
2 node := master
      create: PhaROSTurtleLogger
4     named: '/myTurtleLogger'
      domain: '127.0.0.1'
6     tcpPort: 30303
      xmlRpcPort: 31313
```

Listing 3.4: Running a Topic Subcriber Node

As a result, you'll start seeing velocity set by the publisher displayed on the Transcript. Once you are done, you stop the node by sending it the stop message in a workspace.

3.4 PHAROS Prototyping and Debugging Facilities

3.4.1 Calling a service

ROS allows nodes to call services provided by others through XML RPC. Services are there to make a remote node execute something, or to ask for something punctual. They are opposite to topics, that are instead useful to share data flows.

To call a service from PHAROS we need to write something like the following snippet:

```
1 master := PhaROSMaster url: 'http://127.0.0.1:11311/'.
2 service := master lookupService: '/clear' with: '/myID' .
  service md5sum: 'd41d8cd98f00b204e9800998ecf8427e'.
4 service call.
```

Listing 3.5: Writting a service call

In this example, we first ask the ROS master to lookup for a service. The first parameter is the name under which the service was registered. The second parameter is the identifier of the caller node. In our example, we lookup the '/clear' service registered by the turtle simulator node. This service clears the paths made the turtle motion on the simulator.

Looking up a service is not enought. We needto set up its md5sum before calling it.

Note: An easy way to getting the md5sum of a service is to call with a nil md5sum. The result is an error which message provides all information about the service including the md5sum!

Once we have a reference to a service, we can call it as many time as we need. In the case of our example, we don't need the result. But, in case you need the result, you can get it by evaluating: service response.

3.4.2 Writing a Publisher Node with PhaROSBlockNode

PHAROS provides the PhaROSBlockNode that allows to write nodes quickly, mainly for test and debug purpose. Listing 3.6 shows a snippet of code using this class. It first creates a master object connected to the ROS master (line 2). Then, we create a new node (from line 3 to 7). This node has a name, an ip address to reach the node (the ip address of the machine where PHAROS is installed), two ports one for tcp connection and one for xmlrpc incoming connections.

Using this node from the PHAROS side, we can now send messages to some topic. In this example, we send a command on the /turtle1/command_velocity topic to make the turtle moving.

```
1  | master node |
2  master := PhaROSMaster url: 'http://127.0.0.1:11311/'.
   node := master create: PhaROSBlockNode
4    named: '/myNode'
     domain: '127.0.0.1'
6    tcpPort: 9999
     xmlRpcPort: 6123.
8
   "Assuming the turtlesim is already running. Command: rosrun turtlesim turtlesim_node"
10 "Here an example of sending a command to the turtle"
   node sendTo: '/turtle1/command_velocity' a: [
12   :msg | "The msg object has accessing methods for all the fields in the type."
     msg angular: 1.0.
14   msg linear: -2.0 ]
```

Listing 3.6: Example of minimum code to write a publisher node

Listing 3.6 shows a simple example. A more complete example is provided in Listing 3.7. It shows how to write a node with a block that contains its logic code (from line 8 to 13).

```
1  | master node |
2  master := PhaROSMaster url: 'http://127.0.0.1:11311/'.
   node := master create: PhaROSBlockNode
```

```
4   named: '/myNode'
    domain: '127.0.0.1'
6   tcpPort: 9999
    xmlRpcPort: 6123
8   soul: [
      :myNode | "an object reprensenting /myNode"
10    myNode sendTo: '/turtle1/command_velocity' a: [ :msg |
        msg angular: 1.0.
12      msg linear: -2.0.
]].
14 node execute. "this message send make the soul block executing one time by the current thread"
```

Listing 3.7: A simple reusable publisher node

Line 14 in Listing 3.7 only executes once the soul block of the node. If one need a long lived node, he must create a new process and a loop as show in Listing 3.8. In this listing, the node will send a command to the turtle each seconds in an infinite loop. Then, the node is started in its own thread (line 17).

```
1  | master node |
2  master := PhaROSMaster url: 'http://127.0.0.1:11311/'.
   node := master create: PhaROSBlockNode
4    named: '/myNode'
     domain: '127.0.0.1'
6    tcpPort: 9999
     xmlRpcPort: 6123
8    soul: [ :myNode |
       [  (Delay forSeconds: 1) wait.
10       myNode sendTo: '/turtle1/command_velocity' a: [ :msg |
           msg angular: 1.0.
12         msg linear: -2.0
         ].
14       true
       ] whileTrue
16   ].
   [node execute] fork
```

Listing 3.8: A background publisher node

3.4.3 Writing a Subscriber Node with PhaROSBlockNode

Listing 3.9 shows the code to create a node and add it a delegate block (line 8) that will be executed each time a message is received by this node. Then, on line 13, this new node subscribes to the topic named /turtle1/command_velocity. Now, each message sent on this topic will be received by /myNode and its delegate block will display this message on the transcript.

```
| master node |
master := PhaROSMaster url: 'http://127.0.0.1:11311/' .
node := master create: PhaROSNode
   named: '/myNode'
   domain: '127.0.0.1'
   xmlRpcPort: 6123
   delegate: [ :msg :chn |
      "this block will be executed each time a message arrives on one the topics this node is subscribed to"
      Transcript show: msg value asString.
   ].

"node subscribes to a topic"
node interestedIn: '/turtle1/command_velocity' typedAs: 'turtlesim/Velocity'.
```

Listing 3.9: A node that subscribes to a topic and processes the messages it receives

3.4.4 Writing a Node that is both a Publisher and a Subscriber

Listing 3.10 shows the code to create a node that publish and subscrite information. It is the fusion of the two listings 3.7 and 3.9.

```
| master node |
master := PhaROSMaster url: 'http://127.0.0.1:11311/' .
node := master create: PhaROSNode
   named: '/myNode'
   domain: '127.0.0.1'
   tcpPort: 9999
   xmlRpcPort: 6123
   delegate: [ :msg :chn |
      Transcript show: msg value asString.
   ];
   soul: [ :me |
      me sendTo: '/turtle1/command_velocity' a:[ :msg |
         msg angular: 1.0 ;
          linear: -2.0
      ]
   ].
node interestedIn: '/turtle1/command_velocity' typedAs: 'turtlesim/Velocity'.
```

Listing 3.10: A Node that is both a Subscriber and a Publisher

3.5 Running PHAROS Tests

3.5.1 ROS Environment Variables Set Up

ROS need to be configured for running PHAROS tests. First, stop all your ROS nodes, including the rosmaster. Then, edit the .bashrc file of your linux. Comment or remove the line that exports the ROS_NAME variable. Last, add a line that sets the the ROS_IP variable to 127.0.0.1 as following:

```
export ROS\_IP=127.0.0.1
```

Once the virtual machine is started, you verify in a shell that the `ROS_IP` environment variable is correctly set. Before launching PHAROS and running its tests.

```
viki@ROS$ echo $ROS_IP
127.0.0.1
```

3.5.2 Testing ROS

In case you are in trouble, you can test that ROS is set correclty. Launch ROS using the `roscore` command. The following code shows what you should get:

```
viki@ROS$ roscore
... logging to /home/viki/.ros/log/8ebc0b60-a911-11e2-9979-0800275ddc22/roslaunch-ROS-3480.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://127.0.0.1:51607/
ros_comm version 1.8.10

SUMMARY
========

PARAMETERS
 * /rosdistro
 * /rosversion

NODES

auto-starting new master
Exception AttributeError: AttributeError("'_DummyThread' object has no attribute '_Thread__block'",) in <
      module 'threading' from '/usr/lib/python2.7/threading.pyc'> ignored
process[master]: started with pid [3496]
ROS_MASTER_URI=http://127.0.0.1:11311/
```

```
setting /run_id to 8ebc0b60−a911−11e2−9979−0800275ddc22
Exception AttributeError: AttributeError("'_DummyThread' object has no attribute '_Thread__block'",) in <
    module 'threading' from '/usr/lib/python2.7/threading.pyc'> ignored
process[rosout−1]: started with pid [3509]
started core service [/rosout]
```

You can now easily test that ROS is properly working by sending it an HTTP XML request using cURL in another shell such as:

```
viki@ROS$ curl −X POST −d "<?xml version='1.0'?> <methodCall> <methodName>getSystemState
    </methodName> <params> <param> <value><string>/testId</string></value> </param> </params
    > </methodCall>" http://127.0.0.1:11311 −−header "Content−Type:text/xml"
<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><array><data>
<value><int>1</int></value>
<value><string>current system state</string></value>
<value><array><data>
<value><array><data>
<value><array><data>
<value><string>/rosout_agg</string></value>
<value><array><data>
<value><string>/rosout</string></value>
</data></array></value>
</data></array></value>
</data></array></value>
<value><array><data>
<value><array><data>
<value><string>/rosout</string></value>
<value><array><data>
<value><string>/rosout</string></value>
</data></array></value>
</data></array></value>
</data></array></value>
<value><array><data>
<value><array><data>
<value><string>/rosout/set_logger_level</string></value>
<value><array><data>
<value><string>/rosout</string></value>
</data></array></value>
</data></array></value>
<value><array><data>
<value><string>/rosout/get_loggers</string></value>
<value><array><data>
<value><string>/rosout</string></value>
</data></array></value>
</data></array></value>
```

```
</data></array></value>
</data></array></value>
</data></array></value>
</param>
</params>
</methodResponse>
```

Bibliography

[NDP10] Oscar Nierstrasz, Stephane Ducasse, and Damien Pollet. *Pharo by Example*. Square Bracket Associates, July 2010.

[Par08] Lynne E. Parker. *Handbook of Robotics*, chapter 40. Multiple Mobile Robot Systems, pages 921–941. Springer, 2008.