Making compatible two robotic middlewares: ROS and JdeRobot

Satyaki Chakraborty and José M. Cañas

Abstract—**The software in robotics makes real the possibilities opened by the hardware. In contrast with other fields, robotic software has its own requirements like real time and robustness. In the last years several middlewares have appeared in the robotics community that make easier the creation of robotic applications and improve their reusability. Maybe ROS (Robot Operating System) is the most widespread one, with a wide user and developer community. This paper presents the work towards making compatible two of them, JdeRobot and ROS, both component oriented. A compatibility library has been developed that allows JdeRobot components to directly interoperate with ROS drivers, exchanging ROS messages with them. Two experiments are presented that experimentally validate the approach.**

Index Terms—**robotics middleware, software, ROS**

I. Introduction

MOST of the robot intelligence lies on its software. Once the robot sensor and actuator devices are set, the robot behavior is fully caused by its software. There are many different ways to program in robotics and none is universally accepted. Some choose to use directly languages at a very low level (assembler) while others opt for high-level languages like C, C++ or Java.

Good programming practices are an emerging field in the software engineer area but also in robotics. Several special issues of robotics journals, books on the topic have been published and also specific workshops and tracks have been created inside ICRA and IROS. The [1]Journal of Software Engineering for Robotics promotes the synergy between Software Engineering and Robotics meanwhile the IEEE Robotics and Automation Society (TC-SOFT) has founded the Technical Committee for Software Engineering for Robotics and Automation.

Compared with other computer science fields, the development of robot applications exhibits some specific requirements. First, liveliness and real-time operation: software has to take decisions within a fast way, for instance in robot navigation or image processing. Second, robot software has to deal with multiple concurrent sources of activity, and so the architecture tends to be multitasking. Third, computing power is usually spread along several connected computers, and so the robotic software tends to be distributed. Fourth, the robotic software typically deals with heterogeneous hardware. New sensors and actuator devices continuously appear in the market and this makes complex the maintenance and portability to new robots or devices. Fifth, Graphical User Interface (GUI) and simulators are mainly used for debugging purposes. Sixth, the robotic software should be expansible for incremental addition of new functionality and code reuse.

Mobile robot programming has evolved significantly in recent years. In the classical approach, the application programs for simple robots obtain readings from sensors and send commands to actuators by directly calling functions from the drivers provided by the seller. In the last years, several *robotic frameworks* (SDKs, also named middlewares) have appeared that simplify and speed up the development of robot applications, both from robotic companies and from research centers, both with closed and open source. They favor the portability of applications between different robots and promote code reuse.

Middlewares offer a simple and more abstract access to sensors and actuators than the operating systems of simple robots. The SDK *Hardware Abstraction Layer* (HAL) deals with low level details accessing to sensors and actuators, releasing the application programmer from that complexity.

They also provide a particular software architecture for robot applications, a particular way to organize code, to handle code complexity when the robot functionality increases. There are many options here: calling to library functions, reading shared variables, invoking object methods, sending messages via the network to servers, etc. Depending on the programming model the robot application can be considered an object collection, a set of modules talking through the network, an iterative process calling to functions, etc.

In addition, robotic frameworks usually include simple libraries, tools and common functionality blocks, such as robust techniques for perception or control, localization, safe local navigation, global navigation, social abilities, map construction, etc. They also ease the code reuse and integration. This way SDKs shorten the development time and reduce the programming effort needed to code a robotic application as long as the programmer can build it by reusing the common functionality included in the SDK, keeping themselves focused in the specific aspects of their application.

As developers of JdeRobot framework since 2008 the authors faced a strategic decision: competing with ROS is pointless, instead of that, it is more practical to make compatible JdeRobot applications with ROS framework. From the point of view of the small JdeRobot team, one advantage is to reduce the need of development of new drivers, using instead the ROS ones and focusing the efforts in the applications themselves. Another advantage is the direct use of ROS

Satyaki is with Jadavpur University and José M. is with Universidad Rey Juan Carlos

E-mail: satyaki.cs15@gmail.com, josemaria.plaza@urjc.es

[1]www.joser.org

datasets and benchmarks, which are increasingly common in robotics scientific community. The aim of this paper is to present the current approach to compatibility between ROS and JdeRobot.

Section II gives an introduction to ROS and JdeRobot frameworks. Section III presents two previous approches, while section IV describes the current proposed approach. Two experiments of the compatibility library working are presented in section V. Some conclusions finish the paper.

II. TWO ROBOTIC MIDDLEWARES: ROS AND JDEROBOT

Cognitive robotic frameworks were popular in the 90s and they were strongly influenced by the Artificial Intelligence (AI), where planning was one of the main key issues. One of the strengths of such frameworks was their planning modules built around a sensed reality. A good example was Saphira [12], based on a behavior-based cognitive model. Even though the underlying cognitive model usually is a good practice guide for programming robots, this hardwired coupling often leads the user to problems difficult to solve when trying to do something that the framework is not designed to support.

Modern robotic frameworks are more based on software engineering criteria. Key achievements are (1) the hardware abstraction, hiding the complexity of accessing heterogeneous hardware (sensors and actuators) under standard interfaces, (2) the distributed capabilities that allow to run complex systems spread over a network of computers, (3) the multiplatform and multi-language capabilities that enables the user to run the software in multiple architectures, and (4) the existence of big communities of software that share code and ideas.

One relevant middleware was Player/Stage [3], the de facto standard ten years ago. Stage is a 2D robot simulation tool and Player is network server for robot control. Player provides a clean and simple interface to the robot's sensors and actuators. The client program talks to Player over a TCP socket, reading data from sensors, writing commands to actuators, and configuring devices on the fly. Client programs can be written in any of the following languages: C++, Tcl, JAVA, and Python. In addition, the client program can be run from any machine that has a network connection to the robot or to the machine on which the simulator is running.

Another important example is ORCA [8], [5], an open-source framework for developing component-based robotic systems. It provides the means for defining and developing the building-blocks which can be pieced together to form arbitrarily complex robotic systems, from single vehicles to distributed sensor networks. It uses the ICE communication middleware from ZeroC and its explicit interface definition to exchange messages among the components. It was discontinued in 2009, but was very influential.

Other relevant component-based framework is RoboComp [10], [11] by Universidad de Extremadura. It is open source and also uses ICE communication middleware as glue between its components. It includes some tools based on Domain Specific Languages to simplify the whole development cycle of the components. Most component code is automatically generated from simple and abstract descriptions over a component template. In addition, RoboComp includes a robot simulation tool that provides perfect integration with RoboComp and better control over experiments than current existing simulators.

Other open source frameworks that have had some impact on current the state of the are CARMEN by Carnegie Mellon and Miro by University of Ulm. They also use some component-based approach to organize robotic software using IPC and CORBA, respectively, to communicate their modules. There are also closed source frameworks as well, like Microsoft Robotics Studio or ERSP by Evolution Robotics.

A. ROS

The [2]Robot Operating System (ROS) [9] is one of the biggest frameworks nowadays. It was founded by Willow Garage as an open source initiative and it is now maintained by Open Source Robotics Foundation. It has a growing user and developer community and its site hosts a great collection of hardware drivers, algorithms and other tools. ROS is a set of software libraries and tools that help to build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools simplifies the development of robotics projects. It is multiplatform and multilanguage.

The main idea behind ROS is an easy to use middleware that allows connecting several components, named *nodes*, implementing the robotic behavior, in a distributed fashion over a network of computers using hybrid architecture. ROS is developed under hybrid architecture by *message* passing, mainly in publish-subscribe fashion (*topics* in Figure 1). Message passing of typed messages allows components to share information in a decoupled way, where the developer does not require to know which component sends a message, and vice versa, the developer does not know which component or components will receive the published messages.

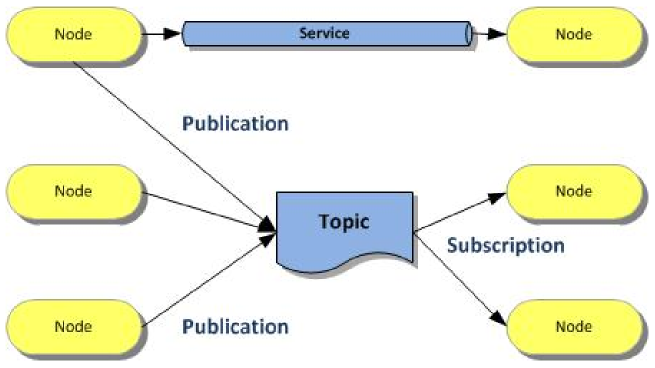


Fig. 1. ROS messages: topics, services

Nodes send and receive messages on topics. A topic is a data transport system based on a subscribe/publish system. One or more nodes are able to publish data to a topic, and one or more nodes can read data on that topic. A topic is typed, the type of data published (the message) is always structured in the same way. A message is a compound data structure. It comprises a combination of primitive types (character strings, Booleans, integers, floating point, etc.) and messages (a message is

[2]http://www.ros.org/

a recursive structure). RPC mechanisms (like *services*) are available as well.

Resources can be reached through a well defined naming policy and a ROS master. Current release is Jade Turtle, the 9th official ROS release. It is supported on Ubuntu Trusty, Utopic, and Vivid.

B. JdeRobot

The JdeRobot platform[3] is a component based framework that uses the powerful object oriented middleware ICE from ZeroC as glue between its components. ICE allows JdeRobot to run in multiple platforms and to have components written in any of the most common programming languages interacting among them. Components can also be distributed over a network of computational nodes and by extension use all the mechanisms provided by ICE as secure communications, redundancy mechanisms or naming services.

The main unit for applications is the component. A component is an independent process which has its own functionality, although it is most common to combine several of these in order to obtain a more complex behavior. There are several types of components, according to the functionality. *Drivers* offer a HAL (Hardware Abstraction Layer) to communicate with the different devices inside the robot (sensors and actuators). The entire configuration needed by the components is provided by its configuration file.

The communication between JdeRobot components occurs through the ICE (Internet Communications Engine) middleware. The ICE communication is based on interfaces and has its own language named *slice*, which allows the developer to define their custom interfaces. Those interfaces are compiled using ICE built-in commands, generating a translation of the slice interface to various languages (Java, C++, Python...). This allows communication between components implemented in any of the languages supported by ICE.

JdeRobot widely uses third party software and libraries (all of them open source) which greatly extends its functionality: OpenCV for image processing; PCL for point cloud processing; OpenNi for the RGB-D support, Gazebo as the main 3D robot simulator and GTK+ for the GUI implementations. Besides, JdeRobot provides its own libraries which give to robotics commonly used functionality for the developing of applications under this framework. It also provides several tools. For instance, it includes `CameraView` tool to show images from any source and includes `kobukiViewer` tool for teleoperating a Kobuki robot and show the data from all its sensors (cameras, laser, encoders).

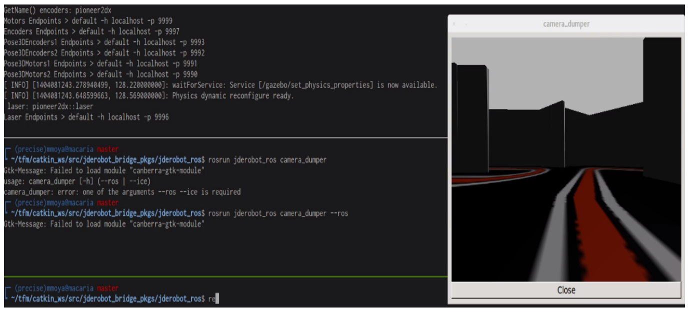
It has evolved significantly since its inception and it is currently at its 5.3 version, which can be installed from packages both in Ubuntu and Debian Linux distributions.

III. Previous works

A. Translator component

In the first approach towards JdeRobot-ROS compatibility we developed a standalone process that translates ROS messages to JdeRobot ICE interfaces and viceversa [1]. This adaptor process is named *jderobot_ros* and allows JdeRobot components to talk to ROS nodes, and allows ROS nodes to communicate with JdeRobot components. It links with both the ICE middleware and the ROS libraries. The translation for every message has to be explicitely coded. The compatibility is intended just for the common sensors and actuators.

In Figure 2 the images from a JdeRobot camera in Gazebo simulator reach the *camera_dumper* ROS node.

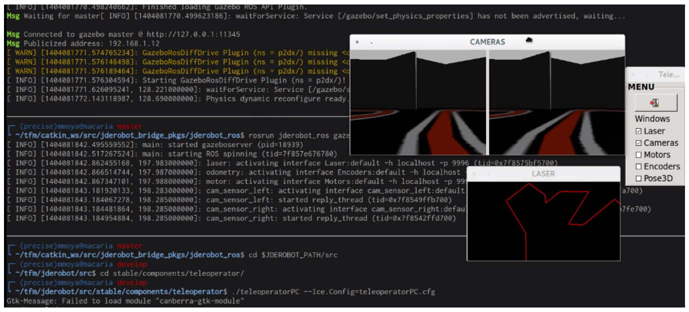

```
GetName() encoders: pioneer2dx
Motors Endpoints > default -h localhost -p 9999
Encoders Endpoints > default -h localhost -p 9997
Pose3DEncoders1 Endpoints > default -h localhost -p 9993
Pose3DEncoders2 Endpoints > default -h localhost -p 9992
Pose3DMotors1 Endpoints > default -h localhost -p 9991
Pose3DMotors2 Endpoints > default -h localhost -p 9990
[ INFO] [1404081243.278940499, 128.220000000]: waitForService: Service [/gazebo/set_physics_properties] is now available.
[ INFO] [1404081243.648599663, 128.569000000]: Physics dynamic reconfigure ready.
 laser: pioneer2dx::laser
Laser Endpoints > default -h localhost -p 9996

┌ (precise)mmoya@macaria master
└ ~/tfm/catkin_ws/src/jderobot_bridge_pkgs/jderobot_ros$ rosrun jderobot_ros camera_dumper
Gtk-Message: Failed to load module "canberra-gtk-module"
usage: camera_dumper [-h] (--ros | --ice)
camera_dumper: error: one of the arguments --ros --ice is required
┌ (precise)mmoya@macaria master
└ ~/tfm/catkin_ws/src/jderobot_bridge_pkgs/jderobot_ros$ rosrun jderobot_ros camera_dumper --ros
Gtk-Message: Failed to load module "canberra-gtk-module"

┌ (precise)mmoya@macaria master
└ ~/tfm/catkin_ws/src/jderobot_bridge_pkgs/jderobot_ros$ re
```

Fig. 2. JdeRobot camera in Gazebo reaches the camera-dumper ROS node

In Figure 3 a ROS Pioneer robot in Gazebo is handled from the JdeRobot *teleoperatorPC* component, that shows images from the robot stereo pair, data from the laser sensor and sends motor commands.


```
[ INFO] [1404081770.498240662]: Finished loading Gazebo ROS API Plugin.
Msg Waiting for master[ INFO] [1404081770.499623186]: waitForService: Service [/gazebo/set_physics_properties] has not been advertised, waiting...

Msg Connected to gazebo master @ http://127.0.0.1:11345
Msg Publicized address: 192.168.1.12
[ WARN] [1404081771.574765234]: GazeboRosDiffDrive Plugin (ns = p2dx/) missing <c
[ WARN] [1404081771.576146498]: GazeboRosDiffDrive Plugin (ns = p2dx/) missing <c
[ WARN] [1404081771.576189464]: GazeboRosDiffDrive Plugin (ns = p2dx/) missing <c
[ INFO] [1404081771.576304594]: Starting GazeboRosDiffDrive Plugin (ns = p2dx/)!
[ INFO] [1404081771.626095241, 128.221000000]: waitForService: Service [/gazebo/s
[ INFO] [1404081772.143118987, 128.690000000]: Physics dynamic reconfigure ready.

┌ (precise)mmoya@macaria master
└ ~/tfm/catkin_ws/src/jderobot_bridge_pkgs/jderobot_ros$ rosrun jderobot_ros gaze
[ INFO] [1404081842.495559552]: main: started gazeboserver (pid=18939)
[ INFO] [1404081842.517267524]: main: starting ROS spinning (tid=7f857e676780)
[ INFO] [1404081842.862455168, 197.983000000]: laser: activating interface Laser:default -h localhost -p 9996 (tid=0x7f8575bf5700)
[ INFO] [1404081842.866514744, 197.987000000]: odometry: activating interface Encoders:default -h localhost
[ INFO] [1404081842.867347101, 197.988000000]: motor: activating interface Motors:default -h localhost -p 99
[ INFO] [1404081843.181920133, 198.283000000]: cam_sensor_left: activating interface cam_sensor_left:default                a700)
[ INFO] [1404081843.184067278, 198.285000000]: cam_sensor_left: started reply_thread (tid=0x7f8549ffb700)
[ INFO] [1404081843.184481864, 198.285000000]: cam_sensor_right: activating interface cam_sensor_right:defau               7fe700)
[ INFO] [1404081843.184954884, 198.285000000]: cam_sensor_right: started reply_thread (tid=0x7f8542ffd700)

┌ (precise)mmoya@macaria master
└ ~/tfm/catkin_ws/src/jderobot_bridge_pkgs/jderobot_ros$ cd $JDEROBOT_PATH/src
┌ (precise)mmoya@macaria develop
└ ~/tfm/jderobot/src$ cd stable/components/teleoperator/
┌ (precise)mmoya@macaria develop
└ ~/tfm/jderobot/src/stable/components/teleoperator$ ./teleoperatorPC --Ice.Config=teleoperatorPC.cfg
Gtk-Message: Failed to load module "canberra-gtk-module"
```

Fig. 3. ROS robot in Gazebo reaches the teleoperatorPC JdeRobot component

B. ROS-ICE Bridge with Template class

In the second approach we tried to avoid the additional translator process [2]. Then we developed a template class to be used in the JdeRobot component that wants to connect to ROS nodes, and to be used in the ROS node that wants to connect to any JdeRobot component. Again, the translation for every message has to be explicitely coded. The compatibility is intended just for the common sensors and actuators.

The *ROS-ICE Bridge* is implemented by using an abstract, template class, which contains an *Ice Proxy*, and also, pointers to *ROS* core components like: ROS-Node, ROS-Publishers and ROS-Subscribers.

1) Sending data from ROS Publisher to JdeRobot component: The workflow for sending data from a ROS interface to a JdeRobot application is described in figure 4. Basically, a derived object from the ROS-Ice class initializes a ROS-Subscriber for the corresponding topic and it also implements

[3]http://jderobot.org

a ROS-callback method. In this function, the derived object must translate the received input into a *slice* format and then send it over to the JdeRobot component, over the Ice-Proxy.

It is worth mentioning that this workflow could be useful if a developer would like to use a graphical user interface from the `Rviz` package. The Rviz package in ROS not only provides features to send data to a backend application, but it also offers a window to visualize real time 3D models of the robot in question.

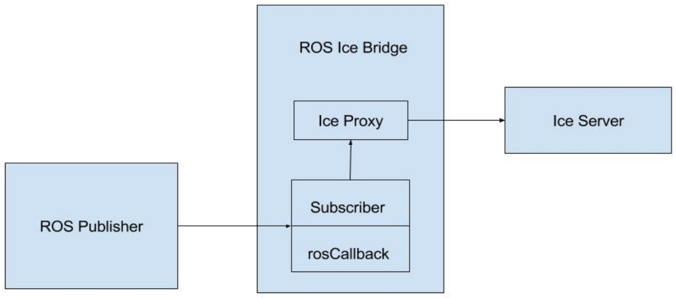


Fig. 4. Workflow of ROS Publisher to ICE component

Another interesting use is getting data on ROS log files from JdeRobot applications. In ROS it is possible to record data in so called *rosbag* files. The framework also provides a way to play-back the recorded data in a synchronized manner and to publish the information on the same topics that were registered. This is possible because at recording time, the timestamp and the channel of the transmitted information, are stored in the *bag* file as well. In this situation, the workflow of the program is as in Figure 4. The only difference is that the ROS Publisher is not developed by the user, instead it is the player in ROS framework which reads the data in the log file.

2) *Sending data from JdeRobot component to ROS applications:* The workflow of the program to receive in a ROS node data from JdeRobot components is as described in Figure 5. The idea is to create a derived object from both the *ROS-ICE* bridge and the "ICE Server" of a random sensor. Once an Ice proxy calls one of its methods, the bridge is supposed to take the input, in slice format, transform it into a ROS message and then publish it on a ROS topic.

This flow of the applications is useful, because many commercial and even industry robots are configured to be controlled and to send information over the ROS framework. Therefore the ROS application could use a JdeRobot driver, if needed, in order to interact with a device.

IV. Compatibility Library

In the current approach we focused only on using ROS nodes (drivers) from JdeRobot applications, as this is the most common and useful scenario. We chose to create a library to be used in JdeRobot components. Using this library the components may talk with ROS drivers too. Regardless the data source, a JdeRobot sensor driver or a ROS sensor driver, the sensor data are mapped to the same local API. The processing side of the application code locally reads the sensor

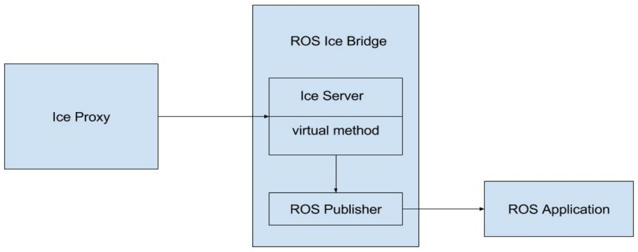


Fig. 5. Workflow of JdeRobot component to ROS application

there. Regardless the data sink, a JdeRobot actuator driver or a ROS actuator driver, the motor commands are mapped from the same local API. The processing side of the application code locally writes the actuator commands there.

The ROS nodes can be used just as the come, without touching at all their code. The library adds the capability of the JdeRobot component to directly talk to ROS nodes if configured to do so.

Current stable version of the JdeRobot middleware follows the ICE server-client architecture. While ICE interfaces help build a stable framework for server-client architectures, using ROS nodes as drivers are advantageous in terms of scalability and reusability. Keeping this idea in mind, we developed a compatibility library that translates ROS messages into local data structures in the client components. The library has been developed to allow the JdeRobot client components (currently supports the *CameraView* component and the *KobukiViewer* component) to communicate both with the ROS drivers as well as their ICE server counterparts. The communication between a typical JdeRobot client component and its ROS driver (via the compatibility library) or ICE server(directly through existing ICE interfaces) is shown in the Figure 6. All the right side of the Figure 6 lies inside the JdeRobot component.

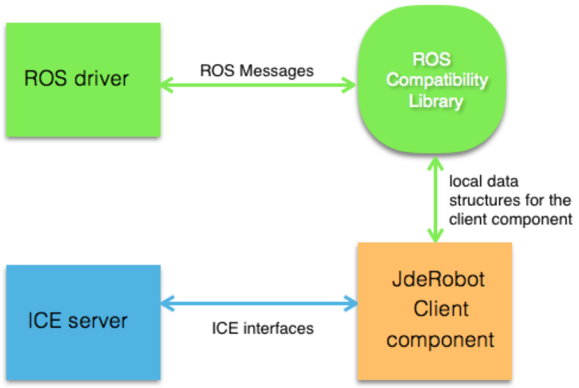

ROS driver
ROS Messages
ROS Compatibility Library
local data structures for the client component
ICE server
ICE interfaces
JdeRobot Client component

Fig. 6. Block diagram showing the communication between a JdeRobot client component with its ROS driver as well as ICE server.

The ROS compatibility library is divided into several segments. Each segment provides methods to translate ROS messages of a particular type of sensor or actuator. For instance, the current implementation contains the following segments for: (1) Image translation (via cvBridge) (2) Laser data translation (3) Encoder data translation (4) Motor data translation. The first segment is used by the CameraView component while all the four segments are simultaneously used by the kobukiViewer component.

The JdeRobot client components are thus modified to support message retrieval from both the ICE servers as well as ROS driver. The user has the option to choose between the ROS driver and the ICE server by setting or resetting a boolean flag in the configuration file of the corresponding client component.

Using ROS drivers with the compatibility library for JdeRobot applications has one significant advantage. As discussed in the previous section, the *ROS-Ice Bridge* architecture uses an intermediate translation of ROS messages into a slice format and then follows the usual ICE server-client architecture to communicate with a JdeRobot component and vice versa. This method (from sending message from ROS publisher to ICE server) involves three steps: (1) sending the message from ROS publisher to ROS subscriber (2) Translating the message into a slice format (3) communicating the information between an ICE client and an ICE server via the ICE interfaces. The same remains true for sending the message from ICE client to ROS subscriber. In order to cut down the overhead, its beneficial to modify the client components by providing them with the ability to receive information directly from either its ICE server or its ROS driver.

V. Experiments

This section contains information about how the ROS compatibility library has been used for the *CameraView* component and the *KobukiViewer* component.

A. *CameraView with a ROS camera node*

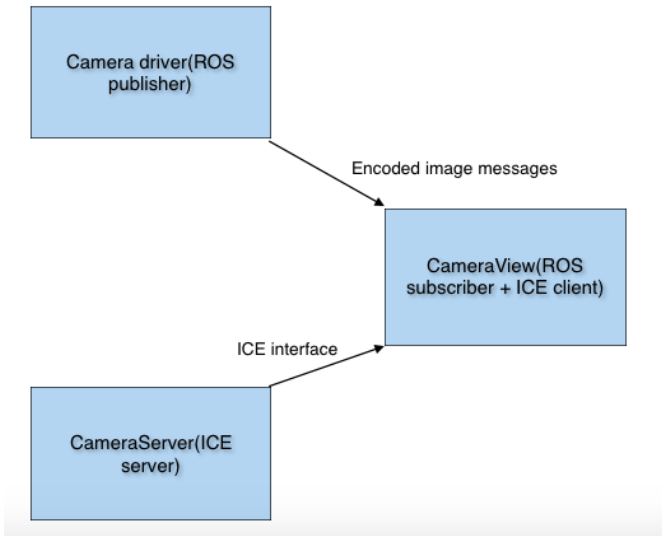
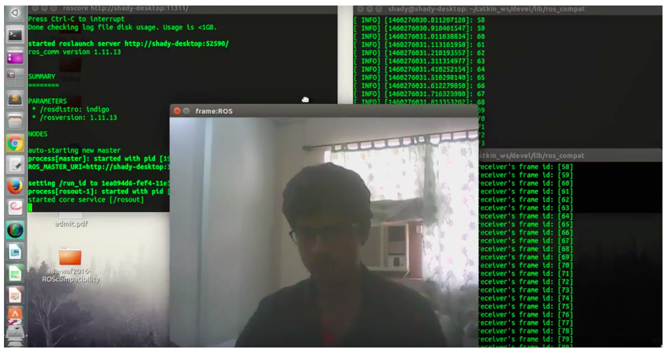

Camera driver(ROS publisher)
Encoded image messages
CameraView(ROS subscriber + ICE client)
ICE interface
CameraServer(ICE server)

Fig. 7. CameraView component may communicate both with a ROS driver as well as an ICE server.


roscore http://shady-desktop:11311/
shady@shady-desktop: ~/catkin_ws/devel/lib/ros_compat
frame:ROS
admit.pdf
aa-waf2016-ROScompatibility

```
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://shady-desktop:52590/
ros_comm version 1.11.13


SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.13

NODES

auto-starting new master
process[master]: started with pid [19
ROS_MASTER_URI=http://shady-desktop:1

setting /run_id to 1ea094d6-fef4-11e5
process[rosout-1]: started with pid [
started core service [/rosout]
```

```
[ INFO] [1460276030.811207128]: 58
[ INFO] [1460276030.910461547]: 59
[ INFO] [1460276031.011638834]: 60
[ INFO] [1460276031.113161958]: 61
[ INFO] [1460276031.210193557]: 62
[ INFO] [1460276031.311314977]: 63
[ INFO] [1460276031.410252154]: 64
[ INFO] [1460276031.510298140]: 65
[ INFO] [1460276031.612279858]: 66
[ INFO] [1460276031.716323980]: 67
[ INFO] [1460276031.813353202]: 68
```

```
eceiver's frame id: [58]
eceiver's frame id: [59]
eceiver's frame id: [60]
eceiver's frame id: [61]
eceiver's frame id: [62]
eceiver's frame id: [63]
eceiver's frame id: [64]
eceiver's frame id: [65]
eceiver's frame id: [66]
eceiver's frame id: [67]
eceiver's frame id: [68]
eceiver's frame id: [69]
eceiver's frame id: [70]
eceiver's frame id: [71]
eceiver's frame id: [72]
eceiver's frame id: [73]
eceiver's frame id: [74]
eceiver's frame id: [75]
eceiver's frame id: [76]
eceiver's frame id: [77]
eceiver's frame id: [78]
eceiver's frame id: [79]
```

Fig. 8. The ROS camera driver is publishing images taken from a USB webcam while the CameraView component is receiving images from the ROS driver and displaying them.

In the current implementation, the *CameraServer* component (the ICE server component) was replaced by a ROS driver. The ROS driver is in fact a ROS node publishing images as ROS messages via *cvBridge*. On the other hand, the *CameraView* component has been modified to act as a ROS subscriber that receives the encoded image messages from the driver. The compatibility library takes care of translating this message into an OpenCV *Mat* object, which is then displayed in the *CameraView* component. Also, along with the images, the frame number or id is also published to keep track of whether the frames are arriving in sequence or not. Figure 8 shows the `CameraView` component being driven by a ROS publisher.

The `CameraView` component is a relatively simple component as in this case only a single node is publishing ROS messages in the driver and in the client component, only a single node is listening to those messages. Hence there is no need of concurrency control or multithreaded spinners. In the next subsection we cover a more challenging problem where the client component receives messages from multiple ROS publishers.

B. *KobukiViewer with a ROS Kobuki robot*

JdeRobot has support for running and testing the koukiRobot or the Turtlebot in a simulated gazebo world. The simulated Kobuki robot (Figure 9) has two cameras, one 2D laser scanner and encoders as sensors, and motors as the only actuators. The ICE servers are implemented as gazebo plugins and the ICE client component (the *KobukiViewer* component) communicates with the servers through different ICE interfaces. The ICE interfaces currently supported by the KobukiRobot simulation in JdeRobot are: 1) Camera Images 2) Laser data 3) Pose3d (Encoder data) 4) Motors.

We thus modified the Gazebo plugins to run ROS nodes in stead of ICE servers. These nodes can act as ROS publisher or ROS subscriber (or both) depending on whether messages are being sent to or received from the client component. The camera driver, laser driver and the pose3d driver only send sensor data as messages to the *kobukiViewer*. Hence in these cases we only need a ROS node to publish these messages in different ROS topics. On the other hand, the motor driver

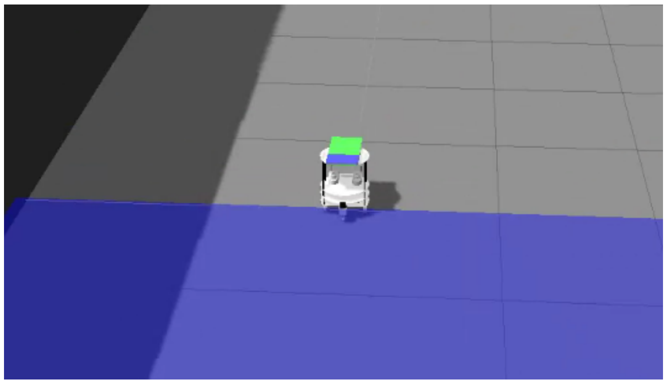
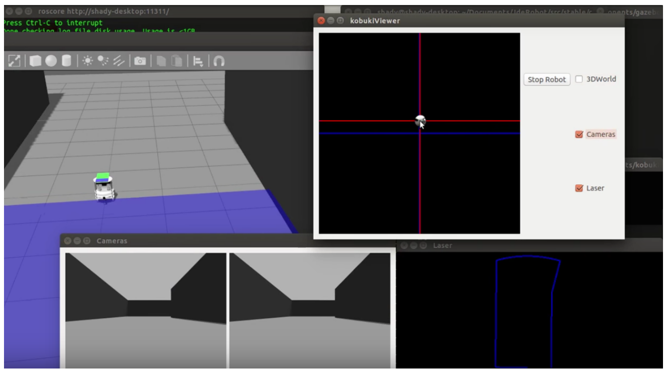


Fig. 9. Kobuki robot simulation in Gazebo.

needs to publish its values as well as receive values from the client component to update its parameters. Hence in this case, the ROS driver needs a bidirectional communication using a ROS publisher node that publishes the actuator data as well as a ROS subscriber node that listens to the client component for updating the values of the actuator.

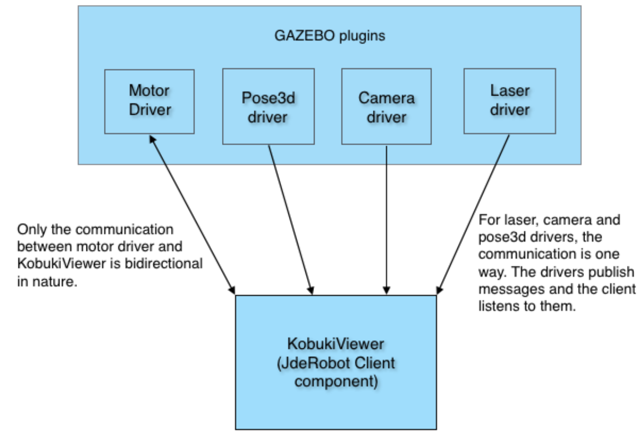


```
roscore http://shady-desktop:11311/
Press Ctrl-C to interrupt
```

Fig. 10. Figure shows KobukiViewer component being run with ROS drivers.

In the *kobukiViewer* component, we have four ROS subscribers subscribing to each of the four ROS topics to receive sensor data and one ROS publisher to publish messages for updating the values of the motors. For each subscriber we implement a callback function where the ROS compatibility library is used to translate the ROS messages into data structures local to the *kobukiViewer* component. The ROS compatibility library is also used in the callback function in the motor driver which listens to the *kobukiViewer* component for messages in order to update the values of the actuator.

In Figure 10, we see the *kobukiViewer* component being run with ROS drivers. The component provides a Qt GUI to manually set the motor values or control which sensor readings to display. In this case, we see the 2D laser scan in the bottom right window and the left and right camera images at that instant in the window just next to it. In order to run the GUI and the ROS functions in parallel, we use a *ros::AsyncSpinner* object to run the ROS callback functions in the background thread. *ros::AsyncSpinner* unlike the single threaded *ros::spin()* function does not conform to abstract Spinner interface. Instead, it spins asynchronously when one calls *start()*, and stops when either one calls *stop()*, *ros::shutdown()* is called, or its destructor is called.



GAZEBO plugins
Motor Driver
Pose3d driver
Camera driver
Laser driver
Only the communication between motor driver and KobukiViewer is bidirectional in nature.
For laser, camera and pose3d drivers, the communication is one way. The drivers publish messages and the client listens to them.
KobukiViewer (JdeRobot Client component)

Fig. 11. Figure shows how the KobukiViewer component communicates with different ROS drivers.

Next, we illustrate a small code snippet to show the implementation of how the JdeRobot client component(s) is(are) modified to support ROS compatibility. The snippet includes two files. First is *roscompat.h*, an header file from the ROS compatibility library which contains methods responsible for message translation.

roscompat.h

```
class ros_compat {
  ...

/* public member functions */
public:
  void translate_image_messages(const sensor_msg::ImagePtr&
       msg, cv::Mat& image);
  void translate_laser_messages(const ros_compat::Num::ConstPtr
       & msg, std::vector<int>& laserdata);
  void translate_pose3d_messages(const ros_compat::Pose3d::
       ConstPtr& msg, std::vector<int>& pose);
  void translate_motor_messages(const ros_compat::Motors::
       ConstPtr& msg, std::vector<int>& motors);
  ...
}
```

In order to illustrate how these functions are used inside the *KobukiViewer* client component, we provide a snippet from the *Sensors.cpp* file from the component.

sensors.cpp

```
#include ‘‘roscompat.h”
...

/* global variables */
ros_compat* rc;
std::vector<int> pose, laserdata, motors;
cv::Mat left_frame, right_frame;
...

/* callback functions */
/* Inside the callback functions */
/* the translation of ROS messages into */
/* local data structures takes place */
/* and the global variables are updated. */
void camera_left_callback(const sensor_msgs::ImageConstPtr&
      image_msg) {
  rc−>translate_image_messages(image_msg, left_frame);
}
void camera_right_callback(const sensor_msgs::ImageConstPtr&
      image_msg) {
  ...
}
void pose3d_callback(const ros_compat::Num::ConstPtr& msg) {
  ...
}
/* so on */


Sensors::sensors(...) {
  ...

  /* ROS intialisation */
  ros::init(argc, argv, "kobukiclient");

  /* Create NodeHandles */
  ros::NodeHandle n_cam_left, n_cam_right, n_laser, n_pose,
        n_motors;

  /* Create ROS subscribers for each ROS topic */
  image_transport::ImageTransport it_left(n_cam_left);
  image_transport::ImageTransport it_right(n_cam_right);
  image_transport::Subscriber left_camera_sub = it.subscribe("
        leftcameratopic", 1000, camera_left_callback);
  image_transport::Subscriber right_camera_sub = it.subscribe("
        rightcameratopic", 1000, camera_right_callback);
  ros::Subscriber laser_sub = n_laser.subscribe("lasertopic", 1001,
        laserCallback);
  /* so on */

  /* Start the AsyncSpinner */
  ros::AsyncSpinner spinner(4);
  spinner.start();
  ros::waitForShutdown();

  ...

}

void Sensors::update() {
   /* update the private datamembers from the global variables */
   /* in stead of using the ICE proxies */
   /* an example is shown for the laser data*/
   /* Sensors::LaserData is a private data member */
   /* whereas laserdata is a global variable updated */
   /* when the callback function is called */

   mutex.lock();
   laserData.resize(laserdata.size());
   for (int i=0; i<laserdata.size(); i++) {
     laserData[i] = laserdata[i];
   }
   mutex.unlock();

   /* same for camera images, pose3d etc. */
   ...
}

/* other public member functions */
...
```

VI. Conclusions

The preliminary work on the third approach to allow compatibility between ROS nodes and JdeRobot components has been presented in this paper. It consists of a compatibility library that extends the capability of the components to connect to ROS nodes exchanging ROS messages. The components may connect with other JdeRobot units using ICE or with ROS units using that library. The ROS message processing is put on a library so any component dealing with the same messages may share it.

The communication is bidirectional, as sensors ('get' operations) and actuators ('set' operations) are supported. The processing for every ROS message has to be explicitely coded. The compatibility is intended just for the common sensors and actuators messages, so the JdeRobot application may use the ROS Hardware Abstraction Layer. The ROS communication side matches the same local API for sensors and actuators that the ICE communication side does, and so, the logic of the JdeRobot component is not changed at all.

The compatibility library has been validated with two experiments connecting two standard JdeRobot applications to ROS nodes. First, the cameraViewer to a ROS node that serves camera images. Second, the kobukiViewer has been connected with a Kobuki robot with motors, cameras, encoders and laser served through ROS.

The main development is focused now is to extend the compatibility library to support a drone robot like 3DR Solo drone and RGBD sensors like Kinect-2. In addition, the compatibility has been tested so far on C++ components, further work is needed to support the same extension on Python JdeRobot components.

Acknowledgment

This research has been partially sponsored by the Community of Madrid through the RoboCity2030-III project (S2013/MIT-2748), by the Spanish Ministerio de Economía y Competitividad through the SIRMAVED project (DPI2013-40534-R) and by the URJC-BancoSantander. Authors also would like to thank Google for accepting JdeRobot on its program GSoC-2015.

References

[1] http://jderobot.org/Mmoya-tfm

[2] http://jderobot.org/Militaru92-colab

[3] Brian P. Gerkey, Richard T. Vaughan, Andrew Howard. *The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems.* In Proceedings of the International Conference on Advanced Robotics (ICAR 2003), pp 317-323, Coimbra, Portugal, June 30 - July 3, 2003.

[4] J. M. Cañas and M. González and A. Hernández and F. Rivas, *Recent advances in the JdeRobot framework for robot programming*. Proceedings of RoboCity2030 12th Workshop, Robótica Cognitiva, pp 1-21, UNED, Madrid, July 4, 2013. ISBN:978-84-695-8175-9

[5] A.Makarenko, A.Brooks, T.Kaupp. *On the Benefits of Making Robotic Software Frameworks Thin.* IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007). Workshop on Evaluation of Middleware and Architectures.

[6] A.Makarenko, A.Brooks, B.Upcroft. *An Autonomous Vehicle Using Ice and Orca.* ZeroC's Connections newsletter, issue 22, April, 2007.

[7] A. Makarenko, A. Brooks, T. Kaupp. *Orca: Components for Robotics.* IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006). Workshop on Robotic Standardization.

[8] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback. *Orca: a component model and repository*. In D. Brugali, editor, Software Engineering for Experimental Robotics. Springer Tracts in Advanced Robotics, 30, p. 231-251, 2007.

[9] 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*. ICRA workshop on open source software. Vol. 3. No. 3.2. 2009.

[10] L. Manso, P. Bachiller, P.Bustos, P. Nuñez, R.Cintas, L.Calderita. *Robo-Comp: A tool-based robotics framework*. In Proceedings of Simulation, Modeling, and Programming for Autonomous Robots: Second International Conference, SIMPAR 2010, Darmstadt, Germany, November 15-18, Springer Berlin Heidelberg, pp 251–262, 2010.

[11] Marco A. Gutiérrez, A. Romero-Garcés, P. Bustos, J. Martínez. *Progress in RoboComp*, Journal of Physical Agents 7(1), pp 39–48, 2013.

[12] Kurt Konolige, Karen Myers. *The saphira architecture for autonomous mobile robots*. In book “Artificial intelligence and mobile robots”, pp 211–242. MIT Press Cambridge, MA, 1998.