Development of an Industrial Robotic Cell Simulation Environment for Safe Human-Robot Interaction Purposes

Supervisors:
Prof. Luca Bascetta
Dott. Matteo Ragaglia

Author:
Omar KHALFAOUI
matricola: 782600

Academic Year 2013/2014
To my family..
Abstract

A simulation model of the MERLIN[1] robotic cell has been developed in order to provide a suitable and highly flexible platform for development and validation of safe Human-Robot Interaction (HRI) control algorithms and strategies. The architecture, developed using V-REP simulator and ROS framework, guarantees a bidirectional communication between the virtual replica of the robotic cell and the real world. This thesis investigates and exposes a methodology that allows to read the main informations from every single robot and sensor of the simulation model, publish these packages of data across ROS network, elaborate them and feed them back to the simulator and/or use them as input for the robots controllers.

Un modello di simulazione della cella robotizzata MERLIN[1] è stato sviluppato per fornire un’adatta e flessibile piattaforma per lo sviluppo e la convalida di algoritmi e strategie di controllo per una Interazione Uomo-Macchina sicura. L’architettura sviluppata, usando il simulatore V-REP ed il framework ROS, garantisce una comunicazione bidirezionale tra la il modello virtuale della cella robotizzata ed il mondo reale. Questa tesi indaga ed espone una metodologia che permette la lettura delle principali informazioni relative ad ogni singolo robot e sensore presenti nel modello simulato, pubblicare questi pacchetti di dati attraverso la rete ROS, elaborarli e rinviarli indietro al simulatore e/o utilizzarli come input per i controllori robot.
4.3.2 Cameras ......................................................... 46
4.3.3 Kinect ......................................................... 48
4.3.4 Rangefinder .................................................. 48
4.4 Final structure ................................................ 50

5 Results and Conclusions ........................................ 51

A Installation and Configuration ................................. 67
A.1 ROS Fuerte Turtle ............................................ 67
A.1.1 Sourcing del setup.sh ....................................... 67
A.1.2 Creating a workspace ....................................... 67
A.1.3 Creating a sandbox directory for new packages ......... 67
A.1.4 Creating our Tesi Package ................................ 68
A.1.5 Building our new Package ................................ 68
A.2 V-REP, configuring ROS plugin ............................. 69

B Examples .......................................................... 73
B.1 V-REP publisher and subscriber ............................ 73
B.2 Control a V-REP object via a ROS node ................... 76
B.3 Exchange of information in both directions .......... 80
B.4 Interrogate V-REP Directly from ROS ................... 82

C Other ............................................................... 85
C.1 Proximity Sensors Calibration .............................. 85
C.2 Built-in standard ROS message types ................... 87

D Nodes .............................................................. 89
D.1 Sensors ........................................................ 89
D.1.1 Proximity sensor (proxSensors.cpp) .................. 89
D.1.2 Kinect (kinect.cpp) ........................................ 91
D.1.3 Node telecamera_0 (telecamera_0.cpp) ............... 93
D.1.4 Node telecamera_1 (telecamera_1.cpp) ............... 95
D.2 Robots ........................................................ 97
D.2.1 robot (robot.cpp) .......................................... 97
D.2.2 YouBotHandles (YouBot.cpp) .......................... 99
D.2.3 IrbHandles (Irb.cpp) ...................................... 101
D.2.4 ComauHandles (Comau.cpp) ........................... 103
D.2.5 NYouBot_PosOrRel (NYouBot_PosOrRel.cpp) ......... 105
D.2.6 NYouBot_PosOrAss (NYouBot_PosOrAss.cpp) ......... 107
D.2.7 NIrb_PosOrRel (NIrb_PosOrRel.cpp) .................. 109
D.2.8 NIrb_PosOrAss (NIrb_PosOrAss.cpp) .................. 111
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.2.9</td>
<td>NComau PosOrRel (NComau_PosOrRel.cpp)</td>
<td>113</td>
</tr>
<tr>
<td>D.2.10</td>
<td>NComau PosOrAss (NComau_PosOrAss.cpp)</td>
<td>115</td>
</tr>
<tr>
<td>D.2.11</td>
<td>NComau PosOrAss (NComau_PosOrAss.cpp)</td>
<td>116</td>
</tr>
</tbody>
</table>
Introduction

Sometimes investigating and/or testing the behavior of a system directly by experimentation might not be feasible. In some cases, inputs and outputs may not be accessible, in others, especially in production systems, the experiment may be too dangerous for the safety of the personnel or simply too costly. Using simulations is generally cheaper and safer than conducting experiments with a prototype of the final product. Simulation, in the robotic field, is a relevant issue. It allows to develop, test and validate control algorithms without having to use the physical machine. Once the control code has been tested and validated and when the simulation results satisfy the fixed goals, it is possible to deploy the developed algorithms directly onto the real robot with no need of further changes.

In the first chapter, we will analyse the problem and state our goal of designing a complete and flexible simulation model of the MERLIN robotic cell on the basis of both Robot Operating System (ROS) and Virtual Robot Experimentation Platform (V-REP). In chapter 2, we will describe in details the structure and design of both ROS framework and V-REP environment, along with their respective features. In chapter 3, we will introduce the real workspace we intend to simulate, describing in detail the sensors and robot systems that compose it. Then we will see how to create and configure a model of this real workspace with V-REP. In the 4th chapter, we will design the communication system through which data flows from the simulator to the real world and vice-versa, using ROS platform. Results and conclusions are discussed in the last chapter with some examples that show how this simulation model can be used.
Chapter 1
Problem analysis and state of the art

Different publications have treated and studied the problem of the design of a hardware/software architecture for on-line safe monitoring and motion planning with the objective to assure safety for workers and machinery and flexibility in production control\(^2\). Where the adopted architecture establishes a bidirectional communication between the robot controller and a virtual replica of the real robotic cell developed using Microsoft Robotics Developers Studio and implemented for a six-dof COMAU NS12 robot. Efficient solutions, for programming and safe monitoring of an industrial robot via a virtual cell, have been designed\(^3\). Very satisfying results have been reached in recognizing variously shaped mobile objects inside the monitored area, using Microsoft Kinect depth sensor, and stops the robot before colliding with them, if the objects are not too small. In real cases, a robotic cell is usually equipped with several robots and sensors which should interact and work in harmony taking into account safety measures when interacting with workers. This thesis discusses a flexible simulation architecture which take into account every information provided by robot systems and sensors available in order to realize a data and sensor fusion.

The main goal of this work is to design and simulate a robotic workspaces taking into account every information provided by robot systems and sensors available inside it in order to realize data and sensor fusion. The collection and elaboration of these data is carried out using the ROS framework. Thanks to its publish and subscribe architecture, ROS gives use the opportunity to develop applications, called nodes, that can be used in other robotic systems, that are able to communicate to each other through message exchange. The workspace, on the other hand, is modelled using V-REP which provides
many tools, features and elaborate APIs that allow the creation of a distributed control architecture.

The robotic cell considered is equipped with two fish-eye RGB cameras, one Microsoft Kinect and a Hokuyo URG 04LX UG01 range finder. A matrix of proximity sensors is mounted on one of the robots for safe human-robot-cooperation control methods. On the other hand, we have one mobile robot Kuka Youbot, and two industrial robots, an ABB IRB 140 and a COMAU SMART SiX.

The two stationary cameras are used to detect the presence of moving objects and/or human operators inside the workspace using blob detection methods. The Microsoft Kinect is used for the same purpose however employing a different image processing method, described later in this paper. The proximity sensors mounted on the ABB IRB 140 are ray-type sensors with a perception field that ranges from 0.020m to 0.080m adequately positioned for safety purposes. The range finder is responsible for detecting moving objects inside the workspace and it works complementarily with the cameras to create data redundancy.

The problem of simulating the robotic cell has been divided into two steps. First, a virtual replica of the workspace to simulate is created using V-REP, then the communication architecture is developed on the basis of ROS framework. Once the data, coming from the simulated robots and/or sensors, are retrieved it is possible to elaborate and use them as input for advanced control algorithms whose output can be fed back to the virtual workspace (creating in this way a flexible closed loop that we can easily manage) or directly to the real robot controllers in the physical workspace.

The Virtual Robot Experimentation Platform (V-REP) balances functionality, consists in 3D robot simulator that concurrently simulates control, actuation, sensing and monitoring. As for the Robot Operating System (ROS) framework, we can rely on a collection of tools and libraries with the objective of simplifying the task of creating complex and robust control strategies across a wide variety of robotic platforms. This work investigates in detail these two platforms integrating them together in an attempt of using the best features of each one to create an advanced, powerful and intuitive simulation and control platform.

Moreover, by merging the V-REP simulator and ROS framework we realize a simulator that can be directly connected with the real workspace. by simply substituting the output node of the control graph with the real workspace robot
controller as will be illustrated later in this paper.
Chapter 2

The ROS-VREP simulation platform

The simulation platform adopted in this project is actually a compound of a Robot Operating System (ROS) and the Virtual Robot Experimentation Platform (V-REP). ROS allows information exchange between the robots and the sensors (populating both the real and the simulated workspace) and it also responsible for controlling the manipulators. On the other hand, V-REP, with its versatile architecture design, is used for testing and simulation purposes.

In the first section of this chapter, we will investigate ROS’s architecture and its structure. while in the second section, we will focus on V-REP’s architecture (based on scene objects and calculation modules) and we will describe, with typical V-REP simulation set-ups, the distributed control methodology directly attached to scene objects associated and based on V-REP scripts.

2.1 ROS Overview

ROS has lately experienced a remarkable success in the robotic field. With its open-source philosophy it is possible to develop code and applications that can be shared and used in other robotic systems without much effort. Originally developed in 2007 by the Stanford Artificial Intelligence Laboratory, with the support of the Stanford AI Robot project, today its development continues at Willow Garage[4], a robotics research institute, with more than 20 institutions collaborating within a federated development model.

As a meta-operating system, ROS offers standard operating system features such as hardware abstraction, low-level device control, implementation of com-
monly used functionalities, message passing between processes, and package management. It is based on a Publish and Subscribe Architecture where processes (called nodes) publish and/or subscribe to specific topics on which information is exchanged in the form of messages. Thanks to the re-utilization concept of ROS, robotics engineers and developers can get the code from the repositories, improve it, and share it again. ROS has released some versions, the latest one being Indigo[5]. In this paper, we are going to use Fuerte because of its stability and compatibility with V-REP.

The ROS architecture is mainly divided into two sections or levels of concepts:

- ROS File system Structure
- ROS Architecture

### 2.1.1 ROS Filesystem Structure

As a meta-operating system ROS manages a filesystem, that is divided into folders containing files that describe their functionalities:

**Packages**: packages are collections of software built and treated as an atomic dependency in the ROS build system, where other packages can use it as first-order or indirect dependency and inherit its functions and features. ROS package might contain a ROS runtime process (node), a dataset, configuration files etc . . .

The goal of packages is to create minimal collections of code for easy reuse. Usually, when we talk about packages, we refer to a typical structure of files and folders. The structure usually looks as follows:

- **bin/**: is the folder containing compiled and linked programs (nodes).
- **/include/**: This directory includes the headers of referenced libraries (see (Fig.2.1).
- **msg/**: this is where we will put non-standard messages developed by us.
- **scripts/**: executable scripts.
- **src/**: this is where the source files of our programs are saved.
- **srv/**: this represents the service (srv) types.
- **CMakeLists.txt**: this is the CMake build file for building software packages.
- **manifest.xml**: this is the package manifest file.
2.1. ROS OVERVIEW

**Manifests:** is the description of a package, a minimal specification of it. It has the significant role of declaring dependencies in a language-neutral and operating-system-neutral manner, license information, compiler flags, and so on. Manifests are managed using a file called manifest.xml.

**Stacks:** a named collection of ROS packages is organized into ROS stacks. The goal of stacks is to simplify the process of code sharing. In ROS, we have numerous standard stacks with different uses. An example is the navigation stack.

**Stack manifests (stack.xml):** it is an equivalent of a normal manifest but for stacks.

**Message (msg) types:** Specifications of user-defined messages, exchanged over ROS topics.

ROS has a list of predefined messages, listed in Tab[C.1] but it is also possible to create our own message types as we will see in the next chapters. Message descriptions are stored in my_package/msg with .msg extension.

A special type of ROS messages is **Header**. Is generally used to communicate timestamped data in a particular coordinate frame.

The structure of the Header message is described as:

- uint32 seq
- time stamp
- string frame_id

**Service (srv) types:** Another way of communication between ROS nodes is by using services. They are specifications of functionalities that can be called in a Client-Server manner.

Services are stored in my_package/srv with .srv extension.

After installing and configuring ROS repositories as described in Appendix[A.1], we will obtain the filesystem structure shown in Fig[2.1].

In the upcoming chapters, we will describe more useful tools and learn how to create our own messages and services.

2.1.2 ROS Architecture

ROS publish and subscribe architecture can be interpreted as a peer-to-peer network of ROS nodes which process information and exchange data among themselves, using messages, through a communication channel called topic.

The basic graph concepts are:
Nodes, processes that elaborate data and exchange information via messages sent/read thanks to the Publish/Subscribe architecture. Usually, a system is composed of several nodes, each one taking care of one or more functionalities. In fact, it is usually better to have many nodes that provide only a single functionality rather than a large node that makes everything in the system.

Messages, a message carries information that a certain ROS node intends to share with other ROS nodes. As we enounced in the previous section, in ROS we have many standard types of messages, and we can also develop our own using these default message types. Each message must have a name to be routed by ROS.

Topics, A node can either publish a message on a certain topic or subscribe to another one in order to read the message published by other nodes. This allows us to decouple the production of information from its consumption. It is important that the name of the topic is unique to avoid problems and confusion between topics with the same name. The use of topics allows ROS nodes to send data in a many-to-many fashion.
Services, services allow a ROS node to act as a Client that requests a particular elaboration to the service itself (acting as a Server). The service will perform the elaboration and will send back an answer to the ROS node. Also, a service must have a unique name.

Master, the last important element of the ROS computation graph is the Master. This provides name registration and lookup for the rest of the nodes. Without the ROS master there will be no communication with nodes, services, messages, and others.

In Fig 2.2 we see a simple example of message exchange between two nodes. As we can notice, the topic plays the role of the communication channel where node_A publish a certain type of message and node_B subscribe to this topic in order to read the message.

![Figure 2.2](image.png)

**Figure 2.2:** ROS Computation Graph. Two nodes can communicate via messages which flow through a communication channel called topic.

At this point we acquired a general knowledge about the ROS architecture and how it works. In the upcoming chapters, we will start to understand the applications of ROS’s tools. In the next section instead we will examine V-REP’s architecture.

### 2.2 V-REP’s Architecture

V-REP is a robotic system simulator designed around a versatile and adaptive architecture. V-REP possesses various relatively independent functions, features, or more elaborate APIs, that can be enabled or disabled as desired. By allowing integrated development environment, a distributed control architecture is obtained: each object/model can be individually controlled via an embedded script, a plugin, a ROS node, a remote API client, or a custom solution. Controllers can be written in C/C++, Python, Java, Lua, Matlab, Octave or Urbi.\[6\]
The V-REP simulator is composed of three central elements as shown in the Fig.2.3. The main elements in V-REP used to build a simulation scene are the so called scene objects.

The following scene objects are the one used in this thesis.

**Joints:** elements that link two or more scene objects together with one to three degrees of freedom. We will see how to control and read data from a joint in the third chapter.

**Dummies:** mainly used for examples and seen as helpers. Dummies are simply points with orientation, or reference frames, that can be used for various tasks.

**Proximity sensors:** these scene objects perform an exact minimum distance calculation within a given detection volume. V-REP offers the possibility to model almost any type of proximity sensor, from ultrasonic to infrared, and so on.

**Rendering sensors:** they are camera-like sensors, allowing to extract complex image information from a simulation scene, for instance colors, object sizes, depth maps, ext...

We will describe and investigate in details the scene objects listed above in the following chapters. The scene objects can be considered as the basic building blocks. They can be combined with each other and form complex systems together with calculation modules and control mechanisms. As we can see in
2.2. V-REP’S ARCHITECTURE

Figure 2.4: the three central elements which compose the V-REP structure.

Fig 2.5: V-REP provides 12 different default types. In addition, V-REP encapsulates several calculation modules that can directly operate on one or several scene objects. These calculation modules include, as shown in Fig 2.5:

- the collision detection module,
- the minimum distance calculation module,
- the forward/inverse kinematics calculation module,
- the physics/dynamics module,
- the path/motion planning module.
These calculation modules can be combined with each other and form complex systems together with scene objects and control mechanisms.

The Dynamic V-REP modules support three different physics engines: The Bullet Physics Library, the Open Dynamics Engine, and the Vortex Dynamics Engine.

Regarding the third part of the V-REP architecture, the simulator offers various means for controlling simulations or even to customizing the simulator itself (see Fig. 2.7). These control mechanisms are divided into two categories:

- Local interfaces: Embedded scripts, Plugins, Add-ons
- Remote interfaces: Remote API clients, Custom solutions, and ROS nodes.

The main advantage of the V-REP control mechanisms is that these methods can easily be used at the same time, and even work hand-in-hand.

V-REP present a modular simulation architecture featured with a distributed control mechanism. This leads to a versatile and scalable framework that fits the simulation needs of complex robotic systems.

One crucial aspect giving V-REP more scalability and versatility is its distributed control approach; it is possible to test configurations with up to several hundreds of robots, by simple drag-and-drop or copy-and-paste actions.

In addition, custom simulation functions can be added via:

![Figure 2.6: V-REP calculation Modules.](image)
2.2. V-REP’S ARCHITECTURE

Figure 2.7: V-REP control mechanisms.

- **Scripts in the Lua language**, *Lua* is a powerful, fast, lightweight, embeddable scripting language designed to support procedural programming. The Lua script interpreter is embedded in V-REP, and extended with several hundreds of V-REP specific commands. Scripts in V-REP are the main control mechanism for a simulation.

- **Extension modules to V-REP (plugins)**, Extension modules allow for registering and handling custom commands or functions. A high-level script command can then be handled by an extension module which will execute the corresponding logic and low-level API function calls in a fast and hidden fashion.

A V-REP simulation is handled when the client application calls a main script, which in turn can call *child scripts*. Each simulation scene has exactly one main script, non-threaded and called at every simulation pass, to handle all default behaviour of a simulation.

Child scripts on the other hand are not limited in number, and are associated with scene objects. Each child script represents a small code or program written in Lua allowing handling a particular function in a simulation.

Child scripts are executed in a cascade way, where each child script is executed by a parent child script (i.e. a parent child script is a child script located further down in the current scene hierarchy). If a child script has no parent child script, then the main script is in charge of executing it.

Child scripts are divided in two categories. They can be either non-threaded or
threaded child scripts:

- **Non-threaded child scripts**: They are pass-through scripts. Which means they return control to the caller directly after each simulation pass they execute. If control is not returned to the calling script, then the whole simulation halts. Non-threaded child scripts are normally called at each simulation step. This type of child script should always be used over threaded child scripts whenever possible. A non-threaded child script is segmented in 3 parts:
  
  - **the initialization part**: executed the first time the child script is called. Usually, initialization code as well as handle retrieval are put in this part.
  
  - **the regular part**: executed at each simulation pass. This code is in charge of handling a specific part of a simulation.
  
  - **the restoration part**: executed one time just before a simulation ends. If the object associated with the child script is erased before the end of the simulation, this part won’t be executed.

- **Threaded child scripts**: They are child scripts that will launch in a thread. This means that a call to a threaded child script will launch a new thread, which will execute in parallel, and then directly return. There can be as many threaded child scripts running as needed. Threaded child scripts can in their turn call non-threaded as well as threaded child scripts. When a non-threaded child script is called from a threaded that is not the main thread, then the child script is said to be running in a thread. Threaded child scripts have several weaknesses compared to non-threaded child scripts if not programmed correctly: they are more resource-intensive, they can waste some processing time, and they can be a little bit less responsive to a simulation stop command. As for non-threaded child scripts, threaded child scripts are segmented in the same 3 parts as well.

### 2.3 Summary

In this chapter, we investigated the ROS architecture, from its publish and subscribe nature to its filesystem structure. Furthermore, we focused on V-REP architecture and on its three central elements: scene objects, calculation modules and control mechanisms. Once we install, configure and understand how ROS and V-REP interact with each other, as described in Appendix A.1 we can start to realize the virtual
2.3. SUMMARY

replica of the real workspace; In the next chapter, we will introduce the robots and sensors we have in MERLIN lab, describe them and model them in V-REP simulator.
Chapter 3

Model of the scene

MERLIN is the Mechatronics and Robotics Laboratory for Innovation at the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano (1) (see Fig. 3.1). Established in 1992 by Professor Claudio Maffezzoni, and thanks to a group of people committed to research, innovation and technology transfer in the fields of mechatronics, robotics and motion control, the laboratory has grown both in international scientific visibility and as a reference point for the regional industrial activities.

In this project, will be using 3 robotic devices: an ABB IRB140, a COMAU SiX industrial robots and a KUKA YouBot mobile robot. For the sensing part we will deploy:

- Two fish-eye RGB surveillance cameras, used for positioning tracking using blob-detection methods.
- Microsoft Kinect for depth mapping.
- Proximity sensors mounted of the IRB140 for safety control.
• Rangefinder.

3.1 Robots

The robots we are considering in this project are: ABB IRB40, COMAU Smart SiX and the KUKA YouBot mobile robots, described here below:

**ABB IRB 140** (11), a compact and very powerful fast 6-axes robot. It handles a maximum of 6kg with a reach of 810mm (to axis 5). It can be floor mounted, inverted or wall mounted at any angle. The robust design with fully integrated cables adds to the overall flexibility. The robot has been installed in MERLIN lab in June 2009 and is used for studies in human-robot interaction control.

**Figure 3.2:**
ABB IRB 140

**COMAU SMART SiX** , designed to handle applications such as light-duty handling and arc welding and Engineered for use with a variety of optional devices; the reduced wrist dimensions enable high capacity orientation in small spaces with a high degree of repeatability. The manipulator is controlled by a COMAU C4G open controller interfaced to an external PC where customized control algorithms can be run. The robot has been installed in MERLIN lab in July 2011 and is used for interaction control and learning by human-demonstration as well.

**Figure 3.3:**
COMAU SMART SiX
KUKA YouBot \[12\], an omni-directional mobile platform with a 5-DOF manipulator which includes a 2-finger gripper. Primarily developed for education and research, it is on its way to become a reference platform for hardware and software development in mobile manipulation. KUKA YouBot enables researchers, developers and robotics students to write their own software, learn from others’ accomplishments and configure the hardware as required or desired.

### 3.2 Sensors

On the other hand, four different types of sensors are considered which can be categorized into image processing and laser scanning sensors.

**Kinect**, a stereocamera based on depth mapping technology able to determine the position of an object in its field of view. Similar to a sonar, Microsoft Kinect calculates the time needed for the laser signal to hit an object and bounce back to its source creating in this way a 3D representation of the object.\[13\]

**Axis 212 PTZ**, an RGB fish-eye indoor surveillance camera. Two of these vision-sensors are used to track the movement of a human worker inside the work space.


**Hokuyo URG 04LX UG01**  
A laser-scanner rangefinder. The light source of the sensor is an infrared laser of wavelength 785nm with laser class 1 safety. Its scanning area is 240 semicircle with maximum radius 4000mm. Pitch angle is 0.36 and the sensor outputs the distance measured at every point (683 steps). Distance measurement is based on calculation of the phase difference, due to which it is possible to obtain stable measurement with minimum influence from objects color and reflectance.\[14\]

**Proximity sensors**  
Last but not least, a matrix of eleven ray-type proximity sensors with a perception field that ranges from 0.020m to 0.080m adequately mounted on the ABB IRB 140 for safety control purposes.

To summarize, our virtual replica of the real workspace will contain the following robots and sensors:

- an ABB IRB 140,
- a youBot KUKA,
- a COMAU SiX robot,
- one Microsoft Kinect RGB-Depth sensor,
- two Axis 212 PTZ cameras,
- one Hokuyo URG 04LX UG01 rangefinder,
- and eleven proximity sensors.

At this point, we can start creating our virtual lab in V-REP.
3.3 MERLIN model in V-REP

Models in V-REP are sub-elements of a scene. They are defined by a selection of scene-objects (Par. 2.2) built on a same hierarchy tree. They can be easily loaded using drag-and-drop operation between the model browser and a scene view or directly from the menu bar [File -> Load model]. V-REP provides a collection of robots and sensor models. Under /Models/components/sensors we can find the models for ABB IRB124 and COMAU SMART SiX, while we can find KUKA YouBot model under /Models/robots/mobile.

It is possible to control the IRB140 and YouBot through the user interface that is displayed during simulation when the robots are selected as it is shown in Fig. 3.9 and 3.10. Robots can be controlled by enforcing direct kinematics (imposing joint angles), by solving inverse kinematics (imposing EE position and orientation) and by using path-planning solvers.

![Figure 3.9: ABB IRB140 V-REP model](image)

![Figure 3.10: KUKA YouBot V-REP model](image)

For what concerns the sensors, Kinect and Hokuyo URG 04LX UG01 are found under /Models/components/sensors. To represent the Axis 212 PTZ camera, we will consider a simple perspective vision-sensor and configure it later on. To add a perspective vision-sensor we go to [Add -> Vision_sensor](image)
Regarding the proximity sensors, a detailed description will be presented in the next section. A representation of the final model is shown in Fig.3.11. At this point, before we start the simulation, we need to configure and initialize our models.

3.3.1 Configuring Proximity sensors

In order to allow the development of safety-oriented control strategies, eleven proximity sensors on the ABB IRB140. To add a proximity sensor have been installed in the scene we go to [Menu bar -> Tools -> Scene object properties], the easiest way to create the remaining sensors is to copy-paste the one we just created.

V-REP offers six types of proximity sensors (see Fig.3.12):

- Ray-type
- Randomized ray-type
- Pyramid-type
- Cylinder-type
- Disk-type
- Cone-type
It is possible to model almost any proximity sensor subtype, from ultrasonic to infrared, and so on. In order to select the sensor subtype, we need to open the proximity sensor properties which are part of the scene object properties dialogue, located at [Menu bar -> Tools -> Scene object properties]. In the scene object properties dialogue, we click the Proximity sensor button to display the proximity sensor dialogue as shown in the example of Fig. 3.13 (the Proximity sensor button only appears if the last selection is a proximity sensor).

Figure 3.12: types of proximity sensor, from left to right: Ray-type, Pyramid-type, Cylinder-type, Disk-type, Cone-type, Randomized ray-type.

The sensors we will be using are laser-subtype proximity sensors that range from 0.020m to 0.080m. The transformation matrices used to position the eleven sensors are reported in Appendix C.1. The final result is shown in Fig. 3.14, 3.15, and 3.16.

Figure 3.14: Proximity sensors distribution. front view
Figure 3.13: Proximity sensor dialogue. Example of a proximity sensor with an ultrasonic subtype

Figure 3.15: Proximity sensors distribution. top view
3.3.2 Configuring the Rangefinder

Rangefinders are devices able to measure the distance between an observer A and a target B.

The rangefinders used in this project employ active methods to measure the distances; The most common methods are: time of flight and frequency phase-shift technologies. The first one exploits the laser technology which operates on the time-of-flight principle by sending a laser pulse in a narrow beam towards the object and measuring the time taken by the pulse to be reflected off the target and returned back to the source. While the frequency phase-shift method measures the phase of multiple frequencies on reflection then solves some simultaneous equations to give a final measure.

In V-REP, rangefinders are categorized into:

- **vision-sensors** based rangefinders: High calculation speed but low precision,
- **proximity-sensors** based rangefinders: High precision calculating the geometric distances but relatively low calculation speed.

**Vision-Sensors Rangefinders:**

At this point, it is important to explain some of the fundamental concepts of vision-sensors. They are very powerful as they can be used in various and flexible ways. They can also be used to display still or moving images from an external application or a plugin\(^1\). Plugins can provide customized image processing algo-

A vision-sensor usually produces two types of images at each simulation step: an RGB image and a depth map. Those two images can be inspected using appropriate API function calls, and iterate over each individual pixel or depth map value. While this approach allows maximum flexibility, it is however troublesome and impractical. Instead, it is much more convenient (and fast!) to use the built-in filtering and triggering capabilities. There are several built-in filters that can be applied to images of a vision sensor, they can be composed in a very flexible way by combining several components. Fig. 3.17 illustrates a simple filter used in the Microsoft Kinect model, it permits to derive the intensity scale from the depth map.

**Figure 3.17:** Filter used by Microsoft Kinect V-REP model to derive the intensity scale map

Each component can perform 4 basic operations:

1. Transfer data from one buffer to another (e.g. original depth image to work image)
2. Perform operations on one or more buffers (e.g. intensity scale work image)
3. Activate a trigger (e.g. if average image intensity > 0.3 then activate trigger)
4. Return specific values that can be accessed through an API call (e.g. return the position of the center of mass of a binary image)

The different types of buffers to which a component can access are illustrated in Fig. 3.18. V-REP has more than 30 built-in filter components that can be combined as needed. In addition, new filter components can be developed through plugins.
Let’s go back to our rangefinders based on vision-sensors, V-REP offers two interesting models, a 3D laser scanner fast and a Hokuyo URG 04LX UG01_Fast shown in Fig. 3.19:

![3D Laser Scanner Fast](image1)

![Hokuyo URG 04LX UG01_Fast](image2)

**Figure 3.19:** Rangefinder based on vision sensors

**3D Laser Scanner Fast**, based on a vision-sensor with a perspective angle equal to 45 degrees, a resolution of 64x64 and minimum and maximum distance of operation, respectively, 00.0500m and 05.0000m. The image processing is done by implementing a filter obtained by applying the following sequence of components:

- **Original depth image to work image**: transfers data from one buffer to another, in this case to work image.
- **Extract coordinates from work image**: extracts the coordinates from
the buffer work image. It is possible to specify the number of points to extract along each axis by double clicking on the component to edit it.

- **Intensity scale work image**: extracts the intensity scale map from the buffer work image.
- **Work image to output image**: transfers this information to the temporary buffer output image.

Another option enabled in this sensor is the ability to ignore the RGB information in order to speed up the calculation. In Fig. 3.20 we can see the result of the test of this sensor; Once the simulation is started a floating view, associated with the vision-sensor of the 3D laser scanner, is created and displays the output buffer or, in this case, Intensity scale map.

![Figure 3.20: Simulation started using as proximity sensor the 3D Laser Scanner Fast based on vision sensor.](image)

**Hokuyo URG 04LX UG01_Fast**, This sensor is also working in perspective mode with an operability angle equal to 120 degrees, and a resolution of 256 x 1 which means it scans along a line as shown in Fig. 3.21. It has a minimum and maximum distance of operability respectively equal to 0.0400 and 5.0000. The image processing in this case is similar to the previous sensor except that in this case the **intensity map scale** component is omitted, in fact during the simulation we don’t have any floating view by default. The filter components are:
3.3. MERLIN MODEL IN V-REP

- Original depth image to work image
- Extract coordinates from work image
- Work image to output image

We can derive the floating view for the Hokuyo by following these four steps:

1. Add the missing component in our filter i.e. Intensity scale work image as second last.
2. select the vision-sensor associated with the Hokuyo URG 04LX UG01 Fast in the scene hierarchy.
3. right-click on the scene and select [Add -> Floating View]
4. right-click inside the floating view that we just created [view -> Associate view with selected vision sensor]

Thus, we obtain the result in Fig. 3.21:

![Figure 3.21: Simulation started using as proximity sensor the Hokuyo URG 04LX UG01 Fast based on vision sensor.](image)

As can be seen in this figure, an area inside the floating view, a black line, indicates the presence of objects/person in a certain distance.

**Proximity-Sensors Rangefinders:**

As we saw in the precious section, V-REP offers a very powerful and efficient way to simulate proximity sensors. They are used to model very accurate Laser
Scanners Rangefinders such as 3D laser scanner (not to be confused with the 3D laser scanner “fast” previously discussed) and Hokuyo URG 04LX UG01 (also not to be confused with Hokuyo URG “Fast” 04LX UG01). Both rangefinders have an associated proximity sensor with laser-subtype. This option has no direct effect on how the rangefinder will operate, it will simply discard some entities from detection that were not tagged as detectable. The only difference between the two rangefinders is in the offset and the range of each proximity sensor associated; the 3D laser scanner has, respectively, 4.9450 and +0.0550, while the Hokuyo URG 04LX UG01 has, respectively, 5.5780 and +0.0220.

In Fig.3.22 and Fig.3.23 we have, respectively, the output of the 3D laser scanner and Hokuyo URG 04LX UG01.

![Figure 3.22: Simulation started using as proximity sensor the 3D Laser Scanner based on proximity sensor.](image)

However, V-REP offers a ROS enabled Hokuyo_04LX UG01 model called Hokuyo_04LX UG01 ROS (although it can be used as a generic ROS enabled laser scanner), based on the existing Hokuyo model that we mentioned in the previous paragraph. It performs instantaneous scans and publishes ROS Laserscan msgs, along with the sensor’s tf (these two messages will be discussed in chapter 4). It has an offset of +0.2300 and a range equal to 6,000.
Let’s see how we can get the information sent by this sensor using ROS; After starting roscore, we launch V-REP and start the simulation. We can notice the following topics generated by the V-REP node:

```bash
rostopic list
/initial_pose
/move_base_simple/goal
/rosout
/rosout_agg
/tf
/vrep/front_scan
/vrep/info
```

We can use the `rviz` tool, a 3D visualization tool for ROS, also used to visualize data from the kinect, and read the data sent by Hokuyo rangefinder.

```bash
rosrun rviz rviz
```

We add a new display type and select TF as its type, we add another Display type but this time as LaserScan, and in option Topic we choose `/vrep/front_scan`. We move the sensor up and down, in the V-REP scene, in order to scan the entire body, we get the following result:

Figure 3.23: Simulation started using as proximity sensor the Hokuyo URG 04LX UG01 based on proximity sensor.
3.3.3 AXIS 212 PTZ cameras Calibration

The transformation matrices which determine the pose of the cameras with respect to the global reference frame of the lab, as show in the Fig 3.11, are respectively

$$T_1 = \begin{pmatrix}
-0.9983 & 0.0558 & 0.0183 & 1.4507 \\
0.0543 & 0.9956 & -0.0770 & 1.6420 \\
-0.0225 & -0.0759 & -0.9969 & 2.8738 \\
0 & 0 & 0 & 1
\end{pmatrix}$$ (3.1)

$$T_2 = \begin{pmatrix}
-0.9981 & 0.0493 & -0.0358 & -0.8468 \\
0.0507 & 0.9979 & -0.0412 & 1.5771 \\
0.0337 & -0.0429 & -0.9985 & 2.8421 \\
0 & 0 & 0 & 1
\end{pmatrix}$$ (3.2)

3.1 and 3.2 are applied to the two vision-sensors using the simSetObjectMatrix API. For this, we need to create a non-threaded child script and associate it to one of the two vision-sensors. The child script to be applied is:

Code 3.1: Cameras calibration

```plaintext
if (simGetScriptExecutionCount() == 0) then
    local hcamera1 = simGetObjectHandle("camera")
    local hcamera2 = simGetObjectHandle("camera0")
    local hpiano = simGetObjectHandle("Plane")

    matx1 = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
    matx1[1] = -0.9983
    matx1[2] = 0.0558
    matx1[3] = 0.0183
    matx1[4] = 1.4507
    matx1[5] = 0.0543
    matx1[6] = 0.9956
    matx1[7] = -0.0770
    matx1[8] = 1.6420
    matx1[9] = -0.0225
    matx1[10] = -0.0759
    matx1[11] = -0.9969
    matx1[12] = 2.8738

    local result = simSetObjectMatrix(hcamera1, hpiano, matx1)

    matx2 = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
    matx2[1] = -0.9981
    matx2[2] = 0.0493
    matx2[3] = 0.0358
    matx2[4] = -0.8468
    matx2[5] = 0.0507
    matx2[6] = 0.9979
    matx2[7] = -0.0412
    matx2[8] = 1.5771
    matx2[9] = 0.0337
    matx2[10] = -0.0429
    matx2[11] = -0.9985
    matx2[12] = 2.8421

    local result2 = simSetObjectMatrix(hcamera2, hpiano, matx2)
end
```

The `simGetObjectHandle` API, used in row 2 and 3, allows us to retrieve an object handle based on its name. In order to apply the transformation matrix to a certain object, we use the `simSetObjectMatrix` API which has the following syntax:

```
number result = simSetObjectMatrix(number objectHandle, number relativeToObjectHandle, table_12 matrix)
```

This API takes as input:

- **objectHandle**: handle of the object,
- **relativeToObjectHandle**: indicates relative to which reference frame the matrix is specified,
- **matrix**: pointer to 12 `simFloat` values (the last row of the 4x4 matrix (0,0,0,1) is not needed).

The x-axis of the orientation component is `(matrix[0], matrix[4], matrix[8])`
The y-axis of the orientation component is `(matrix[1], matrix[5], matrix[9])`
The z-axis of the orientation component is `(matrix[2], matrix[6], matrix[10])`
The translation component is `(matrix[3], matrix[7], matrix[11])`
A double-click on the vision-sensor icon in the scene hierarchy will open the scene object properties dialogue. A click on the Vision sensor button will display the vision sensor dialogue (The Vision sensor button only appears if the last selection is a vision sensor). The dialogue displays the settings and parameters of the last selected vision sensor. In our case, the vision-sensors are configured as follow:

- **Near/far clipping plane**: the minimum / maximum distance, from which the vision-sensor will be able to detect, is respectively 0.01 and 3,
- **Perspective angle**: the maximum opening angle of the detection volume, when the vision-sensor is in perspective mode, is 80,
- **Resolution X/Y**: desired x- / y-resolution of the image captured by the vision sensor. 128x128 In our case.
- **Object size X - Y - Z**: size of the body-part of the vision sensor. This has no functional effect.

At this point we need to configure the filter’s components to apply to the two camera models and prepare them for the blob detection.

So, as we saw before, in order to associate a filter to a camera we need to open the vision sensor dialogue. Click on Show filter dialogue and add the following sequence of filter components:

1. Original image to work image.
2. Intensity scale work image.
3. Blob selection on work image.
4. Original image to work image
5. Work image to output image

Now that the filter is ready, we need to create a floating view for the output image; we right-click on the camera model [Add -> Floating view]. At this point, we right-click on the new floating view [view -> Associate view with selected vision sensor].

Now that our camera is ready, we need to specify which scene object should the camera render. In the example shown in Fig.3.25 we choose to detect a moving KUKA YouBot. To do so, from the vision sensor dialogue, we select the Entity to render, in our case the name of the YouBot in the scene hierarchy: ex. youBot_0. It is important that the object we want to detect has to be Renderable.

When we start the simulation, we obtain the following result (Fig.3.25): As we can see, the vision sensor ignores everything but the YouBot. We will use this approach to observe the movement of a person inside the work space.
3.4 Summary

In this chapter, an exhaustive description of the scene model, from MERLIN lab, adopted in this project was given. Three robots have been considered, two of which are industrial robots, ABB IRB140 and COMAU SMART SiX, while the third is a KUKA YouBot mobile robot. A collection of sensors adequately positioned all over the laboratory categorized mainly into two groups: RGB (or RGB-D) sensors and laser scanners. In the first group we find the Microsoft Kinect depth sensor and two Axis 212 PTZ RGB surveillance cameras, in the second group instead we have a chain of proximity sensors mounted on the IRB140 and a range-finder fixed in a suitable position in the laboratory. The next step was to add and configure these devices in V-REP. Some of these device models already exist in V-REP, others, like the Axis 212 PTZ, were modelled using a vision sensor model and configured appropriately.
At this point, the V-REP model of the MERLIN lab is ready. In the next chapter we will see how to design the communication between V-REP and ROS and how to read data from the robot and sensor models.
Chapter 4

ROS-VREP communication

In this chapter, we will combine ROS and V-REP to obtain an architecture where messages containing sensor signals and robot data can be exchanged between the simulated and the real workspace. Moreover, the architecture’s structure will be further detailed describing its main elements: nodes, topics, message types, robot models and sensor models.

In the first section, we will illustrate the general architecture, partitioned into a robot-subset and a sensor-subset and we will describe the APIs used by ROS nodes in order to collect information from the V-REP simulator.

In the second section, we will analyse the robot-subset which is organized into two classes: 1) A HUB system, in charge of collecting and distributing robot and joint handles to the second class, 2) A Control-nodes class which consists of a cluster of control nodes which read the data sent by the HUB system and treat it as input for some control strategy.

In the third section, we will analyse the sensor-subset which contains a collection of nodes in charge of interrogating the V-REP node in order to retrieve sensed data.

4.1 General structure

As announced in the introduction, for adaptability and expandability reasons, we decided to split our architecture into a robot-subset and a sensor-subset. In the first one, we adopted a HUB architecture where several nodes perform the collection and distribution of basic data needed by the control nodes, while a single HUB node is responsible for nodes coordination and information sharing. In the sensor-subset instead, every node is connected directly to the V-REP simulator as shown in Fig 4.1. In order to process the informations exchanged
between our ROS structure and V-REP simulator, we will exploit some of the APIs provided by the V-REP plugin. More precisely, we will be using the following 4 APIs:

- **vrep_common::simRosGetObjectGroupData**: This API allows us to simultaneously retrieve data from various objects in a V-REP scene. Very helpful in our case as we need to retrieve the handles of all proximity sensors.
  - **Input**
    - *objectType (int32)*: a scene object type\(^1\) or sim\_appobj\_object\_type for all scene objects.
    - *dataType (int32)*: the type of data that is desired\(^2\).
  - **Output**
    - *handles (int32[]):* the object handles
    - *intData (int32[]):* the integer data
    - *floatData (float32[]):* the float data
    - *strings (string[]):* the strings

- **vrep_common::simRosGetObjectHandle**: Retrieves an object handle based on its name.
  - **Input**
    - *objectName (string)*: name of the object.
  - **Output**
    - *handle (int32)*: -1 if operation was not successful, otherwise the handle of the object

- **vrep_common::simRosGetVisionSensorDepthBuffer**: Retrieves the depth buffer of a vision sensor.
  - **Input**
    - *handle (int32)*: handle of the vision sensor,
  - **Output**
    - *result (int32)*: -1 if operation was not successful
    - *resolution (int32[]):* 2 values for the resolution of the image
    - *buffer (float32[]):* the depth buffer data. Values are in the range of 0-1 (0=closest to sensor, 1=farthest from sensor).

\(^1\)For instance: sim\_object\_proximitysensor\_type.
\(^2\)List of all DataTypes:


4.2 ROBOT SUBSET

- vrep_common::simRosGetVisionSensorImage:
  Retrieves the image of a vision sensor.
  
  - Input
    * handle (int32): handle of the vision sensor
    * options (uint8): image options, bit-coded: bit0 set: each image pixel is a byte (greyscale image (MONO8)), otherwise each image pixel is a rgb byte-triplet (RGB8).
  
  - output
    * result (int32): -1 if operation was not successful
    * image (sensor_msgs/Image): the image

At this point we can start describing the various nodes belonging to our architecture.

![Graph structure](image)

**Figure 4.1: Graph structure**

### 4.2 Robot subset

The HUB system is in charge of collecting the handles of each robot together with its respective joints handles in order to collect data regarding the robot state, elaborate them and send the results of this elaboration to the various control nodes as shown in as shown in Fig 4.2.

We will be using the following format to describe a node and a topic: [/node] and /topic. For instance, [/robot] and /position.

---

3 Refer to the ROS documentation:
4.2.1 HUB system

The input node for the HUB system is [/robot]. Its task is to collect all data coming from the robots, including their relative joints handles. These handles are split into three different message types: ComauMsg.msg, IrbMsg.msg and YouBotMsg.msg, each one of these corresponding to a different robot, then the [/robot] node publishes these data on the corresponding topics: /ComauHandles, /IrbHandles and /YouBotHandles. We can consider this process a sort of selection and separation step.

Overall, we have 32 handles; 3 relative to the robots and 29 to their joints. The three messages created by [/robot] have the following structures:

**Code 4.1: ComauMsg.msg**

```c
int32 hComauRobot
int32[7] hComauJoint
```

**Code 4.2: IrbMsg.msg**

```c
int32 hIrbRobot
int32[6] hIrbJoint
```

**Code 4.3: YouBotMsg.msg**

```c
int32 hYouBotRobot
int32[16] hYouBotJoint
```

\(^4\)in V-REP, the YouBot wheels are modelled as rotational joints.
4.2. ROBOT SUBSET

which are published, respectively, in the following topics shown in Fig. 4.3, 4.4 and 4.5:

![Diagram](image)

**Figure 4.3:** /ComauHandles topic, which carries a ComauMsg.msg message

![Diagram](image)

**Figure 4.4:** /IrbHandles topic, which carries a YouBot.msg message

![Diagram](image)

**Figure 4.5:** /YouBotHandles topic, which carries a ComauMsg.msg message

It is possible to verify the successful instantiation and setup of these topics by running the command `rostopic list` from the terminal as shown in Fig. 4.6.

```
root@grendaizer:~/fuerte_workspace/sandbox/tesi# rostopic list
 acompanHandles
 /IrbHandles
 /YouBotHandles
 /initialPose
 /move_base_simple/goal
 /rosout
 /rosout_agg
 /tf
 /vrep/front_scan
 /vrep/info
```

**Figure 4.6:** We can notice the three topics: /ComauHandles, /IrbHandles and /YouBotHandles, generated by the [/robot] node.

The nodes that are connected to [/robot] (and that are granted direct access to the data it publishes), [/Comau], [/Irb] and [/YouBot], as shown in Figures 4.3, 4.4 and 4.5. These three nodes read the messages sent by [/robot], refined the messages’s content and re-publish the refined data on new topics. For instance, the node [/Comau] takes as input a list containing the Comau
handle and its joints handles. By employing the `simRosGetObjectGroupData` API, `/Comau` retrieves the `object name` corresponding to each handle and prepares a message called `vett_JointHandles.msg` to be sent through the topic `/Comau_JointHandles` as shown in Fig. 4.7.

![Figure 4.7](image)

**Figure 4.7:** the `/Comau` node publishes a vector `vett_JointHandles.msg` through the topic `/Comau_JointHandles` to which the two nodes `/NComau_PosOrAss` and `/NComau_PosOrRel` are subscribed.

The message `vett_JointHandles.msg` has the following structure:

**Code 4.4:** vett_JointHandles.msg

```plaintext
JointHandles[] handles
```

it contains a vector `handles` of type `JointHandles`, which in turn is a message type structured in the following manner:

**Code 4.5:** JointHandles.msg

```plaintext
int32 indice
int32 handle
string nome
```

Where, `indice` is the index, `handle` and `nome` are, respectively, the handle and name of the object.

### 4.2.2 Control nodes

The main task of the HUB structure is to collect and separate the handles into 3 message types to be sent to the corresponding control nodes.

In our architecture, we have six control nodes responsible for retrieving every absolute and relative joint position and orientation (i.e. Euler angles) and for publishing them in separate topics.

Let's consider, as an example, the Comau robot once again. The respective control nodes are `/NComau_PosOrAss` and `/NComau_PosOrRel` as shown in Fig. 4.7, which subscribe to `/Comau_JointHandles`, and publish, respectively, the following two messages:
4.3 Sensor Subset

For the sensing part, 4 nodes have been created in order to acquire and process data sensed by the virtualized sensors (see Fig. 4.9). Notice that we do not have a corresponding node for the rangefinder since its V-REP model already publishes a ROS topic with elaborated informations as we saw in the previous chapter.
4.3.1 Proximity sensors

The [/proxSensors] node handles the information sent by the 11 proximity sensors mounted on the IRB 140. It takes as input the handle of each sensor and, using the simRosGetObjectGroupData API, obtains, for each proximity sensor, a measure representing the distance between an obstacle and the robot (if any).

To obtain this behaviour, we feed this API with the following input:

- in objectType=sim_object_proximitysensor_type
- in dataType=13: Retrieves proximity sensor data.

which gives us the following output:

- in intData (2 values): detection state, detected object handle
- in floatData (6 values): detected point (x,y,z) and detected surface normal (nx,ny,nz)

Once this information is obtained, a vett_proxData.msg message is created. It consists of a vector of type proxData (see Code 4.10, 4.11), where for each sensor, the following informations are stored:

- int32 state, detection state
- int32 objectHandle, detected object handle
- string nomeProx, sensor’s name
- float32[3] point, detected point (x,y,z)
As an example, we can study the behavior of the proximity sensor number 20 when it detects the presence of an object as shown in Fig.4.11. Once the object is detected, the sensor start blinking and, as we can see in Fig.4.12, the informations regarding the state, name of the sensor, handle and position of the object detected are published.

![Figure 4.10](image)

**Figure 4.10:** the [/proxSensors] node publishes a vett_proxData.msg vector through the topic /prox_data.

![Figure 4.11](image)

**Figure 4.11:** V-REP side: when the proximity sensor senses the presence of an object, blinks and sends the message vett_proxData.msg through the topic /prox_data.

![Figure 4.12](image)

**Figure 4.12:** ROS side: we can inspect the values sent from the proximity sensor by calling the corresponding topic with the command: rostopic echo /prox_data.
4.3.2 Cameras

Identical to [/telecamera_1], the [/telecamera_0] node (as shown in Fig.4.9) retrieves the handle of the camera using the simRosGetObjectHandle API which takes as input:

- **objectName** (string): name of the object, in our case: camera0.

and gives as output:

- **handle** (int32): -1 if operation was not successful, otherwise the handle of the object.

Once obtained the handle of camera0, this node can extract a sensor_msgs::Image from it using the simRosGetVisionSensorImage API, which take as input:

- **handle** (int32): handle of the vision sensor, obtained in the previous step.
- **options** (uint8) = 0.

and gives as output:

- **result** (int32): -1 if operation was not successful,
- **image** (sensor_msgs/Image): the image.

Then publishes this message, sensor_msgs::Image, through the /Camera_Image_0 topic as shown in Fig.4.13.

Figure 4.13: the [/telecamera_0/1] node publishes a sensor_msgs/Image.msg vector through the topic /Camera_Image_0.

One of the tools used to display sensor_msgs::Image messages is the image_view package. The corresponding command is the following:

```
rosrun image_view image_view image:=<image topic> [image transport type]
```

in our case, we have:

```
rosrun image_view image_view image:=/Camera_Image_0
```
which exports the image shown in Fig. 4.14.

Figure 4.14: Displaying the Sensor_msgs::Image message using the image_view package.

Alternatively the rqt framework can be used to export the images acquired from the vision sensors. It implements several GUI tools in the form of plugins. It is possible to run all the existing GUI tools as dockable windows within rqt having the possibility to manage, simultaneously, all the various windows on the screen at one moment as shown in Fig. 4.15.

Figure 4.15: Displaying the Sensor_msgs::Image message using the rqt_gui package.

To use rqt we simply need to run the following command from the terminal:

```
rosrun rqt_gui rqt_gui
```

We can add a second image_view by clicking [Plugins -> Image View], and we connect each one of the views with one of the topics published by the two cameras, which are /Camera_Image_0 and /Camera_Image_1.
4.3.3 Kinect

This node uses the `simRosGetVisionSensorDepthBuffer` API, which retrieves the depth buffer of a vision sensor, by taking as input:

- **handle (int32)**: handle of the vision sensor, retrieved using `simRosGetObjectHandle`.

and giving as output the following three parameters:

- **result(int32)**: -1 if operation was not successful,
- **resolution(int32[]):** 2 values for the resolution of the image,
- **buffer(float32[]):** the depth buffer data. Values are in the range of 0-1 (0=closest to sensor, 1=farthest from sensor).

As we can see in Fig.4.16, the Kinect model provides two types of data; an RGB image and a depth map image. It is possible to publish these data through ROS as an image message, for the RGB image, or as depth buffer for the depth map.

![Kinect model on V-REP](image)

*Figure 4.16: Kinect model on V-REP. As we can see, this model allows us to display the RGB and Depth map. It is possible to publish the latter through ROS as a depth buffer data with values that range from 0 to 1 (0=closest to sensor, 1=farthest from sensor).*

4.3.4 Rangefinder

As for the rangefinder, once the simulation starts the sensor begins publishing a `/vrep/front_scan` topic (as shown in Fig.4.17) containing a ROS `sensor_msgs/La-
serScan.msg message. This is the same topic we displayed using rviz in the previous chapter (see Fig 3.24).

sensor_msgs/LaserScan.msg has the following structure:

**Code 4.12: sensor_msgs/LaserScan.msg**

```
// Single scan from a planar laser range-finder

Header header // timestamp in the header is the acquisition time of
// the first ray in the scan.
  //
  // in frame frame_id, angles are measured around
  // the positive Z axis (counterclockwise, if Z is up)
  // with zero angle being forward along the x axis

float32 angle_min // start angle of the scan [rad]
float32 angle_max // end angle of the scan [rad]
float32 angle_increment // angular distance between measurements [rad]

float32 time_increment // time between measurements [seconds] - if your
  // scanner is moving, this will be used in interpolating
  position

float32 scan_time // of 3d points
float32 range_min // time between scans [seconds]
float32 range_max // minimum range value [m]
float32 range_max // maximum range value [m]

float32[] ranges // range data [m] (Note: values < range_min or >
  // range_max should be discarded)
float32[] intensities // intensity data [device-specific units]. If your
  // device does not provide intensities, please leave
  // the array empty.
```

---

**Figure 4.17:** /vrep/front_scan topic containing a sensor_msgs/LaserScan.msg message.
4.4 Final structure

The architecture we developed can be represented by the following graph scheme:

![Diagram](Figure 4.18: Final structure of the project)
Chapter 5

Results and Conclusions

The main goal of this thesis was to create a simulation model for the MERLIN lab in order to provide a suitable and highly flexible platform for robot control algorithms/strategies development, testing and control.

Our proposed solution is based on the tools and features provided by the V-REP simulator and the by ROS framework. V-REP, thanks to its versatile architecture design, has been used for the testing and simulation part, while ROS allowed information exchange between the robots and the sensors, populating both the real and the simulated workspace, and it is also responsible for controlling the manipulators.

By combining ROS with V-REP we were able to design a flexible architecture which mainly consists of two subsets: a Robot Subset and a Sensor Subset.

The robot subset is responsible for collecting information regarding the kinematic state of the manipulators, while the sensor subset acquires data from the vision and proximity sensors, elaborate them and show them as video streaming, depth map or point cloud representation. In order to test the entire platform, we need to follow the coming steps:

- Run roscore from the terminal.
- In a new terminal, run V-REP and open the simulated workspace.
- It is possible to start the simulation right away, but in order to read the data from every robot and sensor it is necessary to run the relative nodes in separate terminals.

We will consider the example shown in Fig.5.1 where an operator, walking inside the workspace, is modelled and the two Axis 212 PTZ RGB surveillance cameras are used to track him using blob detection method. As a result, by running the rostopic list command, we obtain the list of topics shown in Fig.5.2. As we ex-
Figure 5.1: Example simulation

Figure 5.2: List of all topics published by the simulated workspace. The green ones are those published by the sensors while the blue ones are those published by the robots.

explained in chapter 4, /Comau_PosOrAss and /Comau_PosOrRel (the same goes for /Irb_PosOrAss, /Irb_PosOrRel, /YouBot_PosOrAss and /YouBot_PosOrRel)
are the topics published, respectively, by [/NComau_PosOrAss] and [/NComau_PosOrRel] which contain the absolute and relative position and orientation of COMAU SMART SiX model’s joints.

In fact, if we run these topics we obtain the following results (see Fig. 5.3).

The example illustrated in this Figure shows how the absolute position and orientation of the Comau robot model joints change. For instance, the joint SIX_joint2 moves from absolute position \((0.5362, -2.2258, 0.4500)\) and orientation \((1.5707, 1.3294, 1.5708)\) to the new absolute position \((0.4567, -2.0341, 0.4519)\) and orientation \((1.5707, -0.9197, 1.5707)\). The same is true for the rest of the joints.

Regarding the relative position and orientation of the same robot, by inspecting /Comau_PosOrRel published by [/NComau_PosOrRel], we obtain the results shown in Fig. 5.4 The relative position and orientation of the Comau model SIX_joint6 changes, at time \(t\), from position \((0.0025, -2.9802, -0.0227)\) and orientation \((1.0853e-06, -5.9005e-07, 2.9802e-07)\) to the new relative position \((0.0025, -4.1723, -0.0227)\) and orientation \((1.0852e-06, -5.9022e-07, 2.9539e-07)\). The same applies to the Irb 140 and YouBot nodes and topics.

As for the sensing part, The Hokuyo URG 04LX UG01 range finder publishes the distance measurements data. these data can be represented as a point cloud using the rviz tool mentioned in chapter 4.
54 CHAPTER 5. RESULTS AND CONCLUSIONS

Figure 5.4: /Coma PosOrRel topic. In this figure, we can see how the relative position and orientation of the SIX_joint6 joint of the Comau robot model changes.

The result shown in Fig.5.5 is obtained by moving the Hokuyo URG 04LX sensor up and down in order to scan the entire workspace. rviz gives us the possibility to choose the decay time which is the duration, in seconds, to keep the incoming points. In this case, the decay time is set at 30 in order to allow us to track the operator through his moving trail as shown in the previous figure.

Figure 5.5: Hokuyo URG 04LX UG01 point cloud of the simulated scene in rviz. In addition the robot shapes highlighted with different colors, we notice the shape and the trail left by the moving operator.

The path followed by the operator’s model is illustrated in Fig.5.6.

As for the blob detection, we configure the filter’s components to apply to the
two camera models as described in chapter[4] which are as follow:

1. Original image to work image.
2. Intensity scale work image.
3. Blob selection on work image.
4. Original image to work image
5. Work image to output image

Now that our cameras are ready, we need to specify which scene object should the cameras render. In this example we choose to detect the operator. To do so, from the vision sensor dialogue, we select the Entity to render, in our case the name of the operator in the scene hierarchy. It is important that the object we want to detect has to be Renderable.

When we start the simulation, we obtain the following result (see Fig.??): As we can see, the position of the operator is detected by the two camera and rendered in their relative floating views. It is possible to publish the relative x and y position of the blob. These informations are published, considering as an example the camera 1, through the topics /vrep/Blob_Camera_1_X and /vrep/Blob_Camera_1_Y which contain the values shown in Fig.5.8 As long as the operator enters the camera’s field of view, [/vrep] starts publishing the
Blobs detection using the two cameras. As we can see, the operator’s relative position is tracked by the two cameras and rendered in their relative floating views.

Figure 5.8: /vrep/Blob_Camera_1_X and /vrep/Blob_Camera_1_Y topics, containing the relative x and y position of the blob with respect to the camera 1.

data regarding the relative position of the operator.

In order to publish these informations, the following non-threaded child script has been associated with camera 1:

Code 5.1: camera 1 non-threaded child script
if (simGetScriptExecutionCount() == 0) then
hcamera = simGetObjectHandle('camera')
camPos = simGetObjectPosition(hcamera, -1)
simExtROS_enablePublisher("Blob_Camera_1_X", 1, simros_strmcmd_get_float_signal, -1, -1, "PosX")
simExtROS_enablePublisher("Blob_Camera_1_Y", 1, simros_strmcmd_get_float_signal, -1, -1, "PosY")
end
result, t0, t1 = simReadVisionSensor(hcamera)
simSetFloatSignal("PosX", t1[5])
simSetFloatSignal("PosY", t1[6])
simHandleChildScript(sim_handle_all_except_explicit)
if (simGetSimulationState() == sim_simulation_advancing_lastbeforestop) then
  -- Put some restoration code here
end

At line 7, we have the instruction that reads the vision sensor. Since that vision sensor has a filter that does blob detection and returns values, we have a 3rd return value: t1. What kind of values are returned in t1 is explained in the blob detection filter dialogue: in this window, we double-click the Blob detection on work image filter to open its dialogue. And so basically:

1. t1[1]=blob count.
2. t1[2]=value count per blob (vCnt).
4. t1[4]=blob orientation
5. t1[5]=blob position x
6. t1[6]=blob position y
7. t1[7]=blob width
8. etc..

As for the kinect depth sensor and the proximity sensors, the results are similar to the ones given in chapter 4.

It is possible to illustrate the obtained results directly on MATLAB. To do this, we need to:

1. record the topic using ROS bagfiles, therefore we need to create the bagfiles directory as shown in Code 5.2.
2. convert the .bag file into a .csv in order to open it with Matlab.

As an example, we will record the topics: /Comau_PosOrAss, /Irb_PosOrAss, /vrep/Blob_Camera_0_X, /vrep/Blob_Camera_0_Y, /vrep/Blob_Camera_1_X, and /vrep/Blob_Camera_1_Y.
CHAPTER 5. RESULTS AND CONCLUSIONS

**Code 5.2:** prepare the bagfiles directory

```bash
roscd tesi
mkdir ~/bagfiles
cd ~/bagfiles
```

The command used to record these topics is:

**Code 5.3:** start recording the topics

```bash
rosbag record -O topics -e /Comau_PosOrAss /Irb_PosOrAss /vrep
/Blob_Camera_0_X /vrep/Blob_Camera_0_Y /vrep/Blob_Camera_1_X /vrep/
Blob_Camera_1_Y
```

Where `topics` is the name of the generated bagfile. At this point we need to convert this file and create the corresponding .csv files by running the following command:

**Code 5.4:** convert the bagfile into six csv files

```bash
for topic in `rostopic list -b topics.bag`; do rostopic echo -p -b topics.bag $topic >bagfile-$topic.csv; done
```

As a result, we obtain the structure shown in Fig. 5.9

![Figure 5.9: The generated .csv files.](image)

By importing the .csv corresponding to the moving operator, we obtain the results shown in Fig. 5.10. Regarding the remaining topics, in order to retrieve the informations about the Irb 140 a Comau smart six joints we assigned to Comau the motion described by the associated threaded child script [D.2.11] and we controlled the Irb using the user interface shown in Fig. 3.9.

Following, the results of the moving operator and the absolute position of each joint of the mentioned robot models.
Figure 5.10: Blob detection displayed in Matlab. We can notice, as expected, the same path designed in V-REP model.

Figure 5.17: Irb joints absolute position
Figure 5.11: Irb Joint 1

Figure 5.12: Irb Joint 2

Figure 5.13: Irb Joint 3
Figure 5.14: Irb Joint 4

Figure 5.15: Irb Joint 5

Figure 5.16: Irb Joint 6
Figure 5.18: Comau Joint 1

Figure 5.19: Comau Joint 2

Figure 5.20: Comau Joint 3
Figure 5.21: Comau Joint 4

Figure 5.22: Comau Joint 5

Figure 5.23: Comau Joint 6
Figure 5.24: Comau Joint 7

The principal advantage brought by this architecture is that, once a control strategy has been developed and tested, it is possible to apply it directly to the real workspace simply by adding to the system a new node interfacing the simulated environment with the physical machine we want to control, as shown in Fig 5.25 where ROS is used as a communication channel between the simulated and the real workspace, sending the control signals and feeding back the simulator with feedback signals. Moreover, when we have a lot of sensors and robotics systems in the simulation scene, it is possible to run the simulation in real-time by exploiting the ROS network\[1]. As a matter of fact, a running ROS system can comprise dozens, even hundreds of nodes, spread across multiple machines. In our case, we can use one machine (named "Machine 1") to run exclusively the simulation engine (without losing computation time in rendering and displaying), and display the result of the simulation on another machine (named "Machine 2") as it is shown in Fig 5.26. This will allow Machine 1 to devote its entire computational capacity to compute the simulation, while Machine 2 is

\[1\] http://wiki.ros.org/ROS/NetworkSetup
Figure 5.26: Using two machines, one as a simulation engine and one for visualization. By splitting these two tasks, it is possible to simulate a large number of robotic and sensor models.

dedicated only on the visualization of the results.

Figure 5.27: Reading data directly from the real workspace and send it to the V-REPL simulation for further analysis.

Not only our architecture can be used to control real robots from the simu-
lation environment but it can also be applied to simulate robots behaviour on the basis of data sensed in the real workspace. As an example, we can read the data sent by a Microsoft Kinect and send it, through ROS, to the simulation in V-REP, as shown in Fig.5.27.

It is of course possible to combine all these situations and create a complete platform not only for simulation but for monitoring and controlling a real workspace, in our case the MERLIN lab, with great flexibility.
Appendix A

Installation and Configuration

A.1 ROS Fuerte Turtle

A.1.1 Sourcing del setup.sh

Once the ROS Fuerte Turtle package is installed\(^1\), in order to have access to the commands it is important to execute the command sourcing for each new shell that we open, unless we add this command to the .bashrc file. The sourcing command is the following:

```
source /opt/ros/fuerte/setup.bash
```

A.1.2 Creating a workspace

The tool provided for the creation of ROS workspace is the rosws\(^2\). The following command will create a new workspace in /fuerte_workspace which extends the set of packages installed in /opt/ros/fuerte:

```
rosws init ~/fuerte_workspace /opt/ros/fuerte
```

Note that rosws is part of the rosinstall package, which is not installed by default. The following command downloads it using the Ubuntu package manager:

```
sudo apt-get install python-rosinstall
```

A.1.3 Creating a sandbox directory for new packages

This directory will contain our project’s package.

\(^1\)http://wiki.ros.org/fuerte/Installation/Ubuntu
\(^2\)http://docs.ros.org/independent/api/rosinstall/html/rosws_tutorial.html
Each time a change is made to the workspace, it is necessary to re-source
```
source /opt/ros/fuerte/setup.bash
```
to confirm that your package path has been set, echo the `ROS_PACKAGE_PATH`
variable.
```
echo $ROS_PACKAGE_PATH
```
it should look something like this:
```
/root/fuerte_workspace/sandbox:/opt/ros/fuerte/share:/opt/ros/fuerte/stacks
```

### A.1.4 Creating our `tesi` Package

To create a new package in our directory `sandbox` we need to go into it:
```
cd ~/fuerte_workspace/sandbox
```
Then create our package:
```
roscreate-pkg tesi std_msgs rospy roscpp
```
`std_msgs`, `rospy` and `roscpp` are the dependencies that enable our package to
support the exchange of messages, coding python and C++.
Now we need to make sure that ROS can find our new package. It is often
useful to call `rospack profile` after making changes to our path so that new
directories will be found:
```
source /opt/ros/fuerte/setup.bash
```
Now we can verify the proper creation of our package by simply trying to position
ourselves in using the command `roscd tesi`.

### A.1.5 Building our new Package

Once we successfully created our `tesi` package, we need to build it using the
`rosmake [package]` command:
```
rosmake tesi
```
At this point, the final structure of our workspace is shown in Fig. [A.1]

- `bin/`: contains compiled files,
A.2 V-REP, Configuring ROS Plugin

Once the workspace is created as shown in Appendix A.1.4, we proceed with configuring the ROS plugin in order to enable the connection between V-REP and ROS.

First, we need to examine whether the our V-REP distribution is able to load ROS's plugin or not. To do this, we run the roscore, and in another terminal window we run V-REP using the command ./vrep.sh.

Once we run the command, we should see something like this:
As we can see in line 3 and 4 of Code A.1, ROS plugins has been loaded successfully!

Now we need to copy the two stacks, vrep and rosBubbleRob from the installation folder V-REP to our workspace ROS.

The vrep stack is needed in order to compile the plugin, but also to write applications connecting to V-REP via ROS.
textttrosBubbleRob instead is the stack of a very simple robot controller that connects to V-REP via ROS.

The structure of our new workspace is shown in Fig. A.2
It is important to respect this structure so that ROS can track the new V-REP packages. We can check this by using the command roscd vrep, if this command gives an error it means that the workspace’s structure is wrong.

At this point we need to install the vrep_common and vrep_plugin packages.
But before doing so, we need to edit some files as shown in Fig. A.3 The command used is the following:

**Code A.2:** Modify the files pointers. originale file is the file lm pointing to, while destination is the one present in the workspace.

```
$ ln -s [original file] [destination]
```

Note that you must delete the file [destination] if it already exists in the workspace. Now in order to build the vrep_common package, we need to navigate to the vrep_common package folder with:

```
$ roscd vrep_common
```

and type

```
make
```

With this, any other ROS package will be able to use the V-REP services and stream messages by just adding the vrep_common package to its dependencies.

Compiling the vrep_plugin is not much more difficult: we need to navigate to the vrep_plugin package folder with:
A.2. V-REP, CONFIGURING ROS PLUGIN

Figure A.2: Struttura del workspace ROS con plugin vrep

```
Fuerte workspace
```
```
        vrep
        rosBubbleRob
tesi

        bin
        build
        CMakeLists.txt
        include
        launch
        manifest.xml
        msg
        src
        srv
```

**Figure A.2:** Struttura del workspace ROS con plugin vrep

```
$vrep_{plugin}/include/v_repConst.h$ 
$VREP\_INSTALL\_DIR/programming/include/v_repConst.h$
```
```
$vrep_{plugin}/include/v_repTypes.h$ 
$VREP\_INSTALL\_DIR/programming/include/v_repTypes.h$
```
```
$vrep_{plugin}/include/v_repLib.h$ 
$VREP\_INSTALL\_DIR/programming/include/v_repLib.h$
```
```
$vrep_{plugin}/src/v_repLib.cpp$ 
$VREP\_INSTALL\_DIR/programming/common/v_repLib.cpp$
```

**Figure A.3:** The files that we need to modify before building the plugins.
The command used is Code [A.2](#)

```
$ roscd vrep_{plugin}$
```

and type

```
$ make$
```

The plugin should have been generated in the \texttt{vrep\_plugin/lib} folder. We need to copy and paste the created plugin to the V-REP installation folder. Make
sure that the name is `libv_repExtRos.so`.

Once you have the ROS plugin in place, open a terminal and start the ROS master with:

```
$ roscore
```

And in a new terminal, we need to move to the V-REP installation folder and start V-REP using the following command:

```
$ ./vrep.sh
```

At this point, if ROS plugin has been successfully installed, we will have two running nodes `/rosout` and `/vrep`. We can check this by using the `rosnode` tool which allows us to list the active nodes as shown in the following Code. [A.3]

```
$ rosnode list
/rosout
/vrep
```

Now we have all ROS services (18) which allow communication and information exchange with V-REP. To list these services, we can use the `rosservice` tool as shown in the following Code. [A.4]

```
$ rosservice list
/rosout/get_loggers
/rosout/set_logger_level
/vrep/get_loggers
/vrep/set_logger_level
/vrep/simRosAddStatusbarMessage
/vrep/simRosAuxiliaryConsoleClose
...
/vrep/simRosStopSimulation
/vrep/simRosSynchronous
/vrep/simRosSynchronousTrigger
/vrep/simRosTransferFile
```

The `vrep` node publishes a `/vrep/info` topic where travels a message containing information regarding the simulation. The structure of this message is as follows:

```
$ cat VrepInfo.msg
std_msgs/Header headerInfo
std_msgs/Int32 simulatorState
std_msgs/Float32 simulationTime
std_msgs/Float32 timeStep
```
Appendix B

Examples

In this chapter, we will see some how to create the publisher / subscriber connection between V-REP and ROS.

B.1 V-REP publisher and subscriber

Let’s see how we can structure a code that allows the publication and subscription, of two objects present in the same V-REP scene, using ROS.

In this first example, we will do so directly from V-REP side, using a Lua script.

While ROS services in V-REP are enabled as soon as V-REP starts (as we saw in the previous chapter), ROS topic publishing and data streaming happens only on demand and only while the simulation is running.

V-REP offers and enabling and a disabling functions for each publishing and subscribing task.

When we need to activate/deactivate data streaming via a Lua script, then we need the following two functions:

• simExtROS_enablePublisher:

   Enables a publisher (i.e. V-REP will be streaming data on a specific topic).
   At simulation end, all publishers are automatically disabled.
   
   – Lua parameters:
     • topicName: the desired topic name
     • queueSize: the desired queue size
     • streamCmd: the desired type of data to stream\footnote{http://www.coppeliarobotics.com/helpFiles/en/rosPublisherTypes.htm}
     • auxInt1: the first enabling parameter.
     • auxInt2: the second enabling parameter

\footnote{http://www.coppeliarobotics.com/helpFiles/en/rosPublisherTypes.htm}
* **auxString**: the third enabling parameter
* **publishCnt**: the number of times you wish to publish the data, before going to sleep.

- **Lua return values:**
  * **effectiveTopicName**: the effective topic name that will be used to stream the desired data, or nil if there was an error.

- **simExtROS_disablePublisher:**
  Disables a publisher (i.e. V-REP will stop streaming data on a specific topic) previously enabled with **simExtROS_enablePublisher**. At simulation end, all publishers are automatically disabled.

  - **Lua parameters:**
    * **topicName**: the topic name previously returned by **simExtROS_enablePublisher**.

  - **Lua return values:**
    * **referenceCounter**: the value of the reference counter. If the counter is >0, then the publisher is still enabled. If the counter is 0, then the publisher was just disabled. If referenceCounter is -1, there was an error.

When we need to activate/deactivate topic subscription via a Lua script, then we need the following two functions:

- **simExtROS_enableSubscriber:**
  Enables a subscriber (i.e. V-REP will be listening to streaming data on a specific topic). Subscribers can only be enabled while simulation is running. At simulation end, all subscribers are automatically disabled.

  - **Lua parameters:**
    * **topicName**: the topic name to listen to
    * **queueSize**: the desired queue size
    * **streamCmd**: the desired type of data to stream
    * **auxInt1**: the first enabling parameter.
    * **auxInt2**: the second enabling parameter
    * **auxString**: the third enabling parameter

  - **Lua return values:**
    * **subscriberID**: a subscriber ID, or -1 in case of an error. The subscriber ID is needed to disable that subscriber with **simExtROS_disableSubscriber**.

\[\text{http://www.coppeliarobotics.com/helpFiles/en/rosSubscriberTypes.htm}\]
• **simExtROS\_disableSubscriber:**
  Disables a subscriber (i.e. V-REP will stop listening to a specific topic) previously enabled with **simExtROS\_enableSubscriber**. At simulation end, all subscribers are automatically disabled.

  - **Lua parameters:**
    * **subscriberID:** the subscriber ID previously returned by **simExtROS\_enableSubscriber**.
  - **Lua return values:**
    * **result:** false if there was an error.

As an example, let’s see how we can publish the pose of an object *cube* and read this pose by another object *sfera* as shown in Fig. B.1.

**Figure B.1:** Example 1: `/vrep/PosaOggetto` is the name of the topic through which the pose message travels.

Once created these two objects, *cubo* and *sfera*, on V-REP we need to associate a **non-threaded child script** to one of these two objects and insert the following code:

**Code B.1:** child script controlling the publication and subscription in topic `/vrep/PosaOggetto`

```lua
if (simGetScriptExecutionCount() == 0) then  // initialization phase
  local hCubo = simGetObjectHandle('cubo')
  local nomeTopicEff = simExtROS_enablePublisher('PosaOggetto', 1,
    simros_cmd_get_object_pose, hCubo, -1, '')

  local hSfera = simGetObjectHandle('sfera')
  local subSfera = simExtROS_enableSubscriber(nomeTopicEff, 1,
    simros_cmd_set_object_pose, hSfera, -1, '')

  ...
```
In this example, we considered as publisher the object \textit{cubo} and as subscriber the \textit{sfera}. So as a final result, the \textit{sfera} should chase the \textit{cubo}. The activation of the connection is made only once, so the corresponding code is inserted in the initialization phase part. As shown in Code B.1.

As we can see from the Code B.1 the object \textit{cubo} publishes its position via the command \texttt{simExtROS\_enablePublisher} using the variable \texttt{simros\_strmcmd\_get\_object\_pose}. \textit{PosaOggetto} is the name given to the topic. The object \textit{sfera} is registered as a subscriber via the command \texttt{simExtROS\_enableSubscriber} (line 6 Code. B.1) using the variable \texttt{nomeTopicEff}, which is the actual name of the topic generated by the publisher.

As we can see from the Fig B.1 the \textit{/vrep} node publishes and, at the same time, receives messages from the topic \textit{/vrep/PosaOggetto}. Running the command to display active topics we get the following result:

```
rostopic list
/rosout
/rosout\_agg
/tf
/vrep/info
/vrep/PosaOggetto
```

\textbf{B.2 Control a V-REP object via a ROS node}

In the previous example, we saw how we can create two objects, a publisher and a subscriber, directly from V-REP. In this example, however, we will see how we can do the same thing but the publisher, in this case, will be a ROS node. The structure of this example is shown in Fig. B.2.

As has been stated in the previous chapters, the communication between multiple nodes takes place through the exchange of messages (msg) and/or through services (srv). In this example, we focus only on the first method. These files are stored in \texttt{tesi/msg}.
For the purposes of our example, we want to use a `/geometry_msgs/Point.msg` message having the structure shown in Code.

**Code B.3:** `/geometry_msgs/Point.msg`

```c
float64 x
float64 y
float64 z
```

To use this message, we need to copy it into our workspace, specifically under the message folder `tesi/msg`:

```bash
cd tesi/msg
roscp geometry_msgs/Point.msg Point.msg
```

At this point, if it’s not been done yet, we need to prepare the `CMakeLists.txt` so that ROS can convert msg files into C++ sourcecode. To do so, we open `CMakeLists.txt` file and remove the `#` symbol from the following line:

**Code B.4:** `CMakeLists.txt`, remove `#`

```c
#rosbuild_genmsg()
```

Now that our msg is ready, we can proceed with the creation of the Publisher that in this example that we will name, in this example, `talker`. We move to `tesi/src` and create our file:

```bash
cd tesi/src
$ gedit Posizione.cpp
```

and insert the following code:

**Code B.5:** `code Posizione.cpp`

```c
#include "ros/ros.h"
#include "geometry_msgs/Point.h"
#include <sstream>
```
int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker");
    ros::NodeHandle n;
    ros::Publisher chatter_pub =
    n.advertise<geometry_msgs::Point>("chatter", 1000);
    ros::Rate loop_rate(10);
    int count = 0;
    while (ros::ok())
    {
        // message type to be published
        geometry_msgs::Point msg;
        // the new coordinates to be assigned to the V-Rep object
        msg.x=0.5;
        msg.y=0.7;
        msg.z=0.8;
        ROS_INFO("Contatore: %d", count);
        // publication of msg
        chatter_pub.publish(msg);
        ++count;
    }
    return 0;
}

Before compiling, we need to warn ROS about the creation of this new node. So we need to update the CMakeLists.txt by adding the following lines:

**Code B.6: CMakeLists.txt**

```bash
...rosbuild_add_executable(Posizione src/Posizione.cpp)
```

At this point, we can compile our package with the rosmake tesi command.

Now, at the V-REP side, we add a new object in the scene and call it listener. We associate a non-threaded child script to this object and with the following code:

**Code B.7: non-threaded child script associated with the object cubo**

```bash
if (simGetScriptExecutionCount()==0) then
    local hSfera=simGetObjectHandle('listener')
    local subSfera=simExtROS_enableSubscriber('/chatter',1,
        simros_strmcmd_set_object_position,hSfera,-1,'')
end

simHandleChildScript(sim_handle_all_except_explicit)
```

The type of data that the listener will be looking for is simros_strmcmd_set_object_position (or /geometrymsgs/Point.msg). At this point, we can test our example (remember that roscore has to be started before V-REP in order to load correctly the ROS plugins).

As we can see in Fig. B.2, the node talker (the name given to the node by Posizione.cpp, line 8 Code B.5 created in ROS, publishes the information
about the new location in the form a `/geometry_msgs/Point` message through the `/chatter` topic to which the node `/vrep` is registered and assigns this new position to the object `listener` in the scene.

We can verify the creation con the new topic using the command `rostopic list`:

<table>
<thead>
<tr>
<th>Code B.8: List of published topics. Example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rostopic list</td>
</tr>
<tr>
<td>/rosout</td>
</tr>
<tr>
<td>/rosout_agg</td>
</tr>
<tr>
<td>/tf</td>
</tr>
<tr>
<td>/vrep/info</td>
</tr>
<tr>
<td>/chatter</td>
</tr>
</tbody>
</table>


B.3 Exchange of information in both directions

It is obviously possible to have an exchange of information in both directions, where both nodes are at the same time Publishers and Subscribers as shown in Fig B.3.

In this case, we have two nodes: a V-rep node and a ROS node that we will call talker. Both nodes publish and subscribe to only one topic as shown in the Figure above. V-rep node publish, through the topic /posaT, the pose of the object cuboT. This message is received by the node talker which generate a new pose relative to this one, and publish it through the topic /posaL. As the node vrep subscribes to the topic /posaL, the new pose is assigned to the object cuboL.

Similarly to the previous example, we need to copy the /geometry_msgs/PoseStamped under tesi/msg:

```
$ roscp geometry_msgs PoseStamped.msg PoseStamped.msg
```

The vrep node is similar to that created in the previous example. The V-REP scene will have two objects, the first, called cuboT which we want to publish its pose. The second, called cuboL, to which a new pose will be assigned by the node talker.

Once we create these two objects in V-REP, we associate a non-threaded child script to cuboT containing the following code:

Code B.10: Example 3, non-threaded child script associated to cuboT

```plaintext
if (simGetScriptExecutionCount() == 0) then
    local hCuboT = simGetObjectHandle('cuboT')
    local pubCuboT = simExtROS_enablePublisher('/posaT', 1,
        simros_strmcmd_get_object_pose, hCuboT, -1, '')
    local hCuboL = simGetObjectHandle('cuboL')
    local subCuboL = simExtROS_enableSubscriber('/posaL', 1,
        simros_strmcmd_set_object_pose, hCuboL, -1, '')
end

simHandleChildScript(sim_handle_all_except_explicit)
-- Put your main code here
if (simGetSimulationState() == sim_simulation_advancing_lastbeforestop) then
    -- Put some restoration code here
end

As we can notice from Code B.10, the pose of the object cuboT, described by the variable `simros_strmcmd_set_object_pose`, is published in the topic `/posaT`. The object cuboL instead, subscribes, through vrep node, to the `posaL` topic published by the talker. To do so, we used the variable `simros_strmcmd_set_object_pose` as an argument of `simExtROS_enableSubscriber`.

The talker node is created using the following C++ program (to be saved under ..../tesi/src):

Code B.11: Example 3, talker.cpp

```plaintext
#include "ros/ros.h"
#include "geometry_msgs/PoseStamped.h"
#include <sstream>

geometry_msgs::PoseStamped msgCuboT;

void chatterCallback(const geometry_msgs::PoseStamped::ConstPtr & msg)
{
    msgCuboT.pose.position.x = msg->pose.position.x + 1.5;
    msgCuboT.pose.position.y = msg->pose.position.y + 1.5;
    msgCuboT.pose.position.z = 0.05;
    msgCuboT.pose.orientation.x = 0;
    msgCuboT.pose.orientation.y = 0;
    msgCuboT.pose.orientation.z = 0;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker");
    ros::NodeHandle n;
    ros::Subscriber subCuboT = n.subscribe("posaT", 1000, chatterCallback);
    ros::Publisher pubCuboL = n.advertise<geometry_msgs::PoseStamped>("posaL", 1000);
    ros::Rate loop_rate(10);
    int count = 0;
```
B.4 Interrogate V-REP Directly from ROS

In this example, we will see how it is possible to create, directly on ROS, a publisher node and to make sure the V-REP node reads a topic published by it.

As shown in Fig. B.4, we will try to control an object, called sfera, in V-REP. The non-threaded child script associated to this object is the following:

```cpp
while (ros::ok())
{
   pubCuboL.publish(msgCuboT);
   ros::spinOnce();
   loop_rate.sleep();
   ++count;
}
return 0;
```

Now we have to warn ROS for the creation of this new node by adding the following line in CMakeLists.txt file:

```bash
rosbuild_add_executable(talker src/talker.cpp)
```

and we re-build the package using the command rosmake tesi.

Once the two nodes, vrep and talker, are executed we can notice how the cuboL assumes a certain position at a fixed distance from cuboT. Moving this last in the scene, the object cuboL chases it maintaining the same distance between the two.

**Figure B.4:** Example 4: create ROS and V-REP nodes directly from ROS.
B.4. INTERROGATE V-REP DIRECTLY FROM ROS

**Code B.12: child script associate to Example 4**

```plaintext
if (simGetScriptExecutionCount() == 0) then
    local hsfera = simGetObjectHandle("sfera")
    -- Check if the required ROS plugin is there:
    moduleName = 0
    moduleVersion = 0
    index = 0
    pluginNotFound = true
    while moduleName do
        moduleName, moduleVersion = simGetModuleName(index)
        if (moduleName == 'Ros') then
            pluginNotFound = false
        end
        index = index + 1
    end
    result = simLaunchExecutable('controllo', hsfera, 0)
end

simHandleChildScript(sim_handle_all_except_explicit)
if (simGetSimulationState() == sim_simulation_advancing_lastbeforestop) then
    -- Put some restoration code here
end
```

From ROS side, we create a control node, called *controllo*, as follows:

**Code B.13: Example 4, controllo.cpp**

```plaintext
#include <stdio.h>
#include <stdlib.h>
#include "ros/ros.h"
#include <sstream>
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"
#include "geometry_msgs/Point.h"
#include "vrep_common/simRosEnablePublisher.h"
#include "vrep_common/simRosEnableSubscriber.h"
#include "vrep_common/simRosGetObjectHandle.h"

geometry_msgs::Point msg;
bool simulationRunning = true;
float simulationTime = 0.0f;

void infoCallback(const vrep_common::VrepInfo::ConstPtr & info)
{
    simulationTime = info->simulationTime.data;
    simulationRunning = (info->simulatorState.data & 1) != 0;
}

int main(int argc, char **argv)
{
    int handleOfTheObject;
    if (argc >= 2)
    {
        handleOfTheObject = atoi(argv[1]);
    }
    else
    {
        printf("Indicate following arguments: 'handleOfTheObject'!\n"");
        sleep(5000);
        return 0;
    }

    _argc = 0;
    char ** _argv = NULL;
    ros::init(_argc, _argv, "nControllo");
    if (!ros::master::check())
        return 0);
```
ros::NodeHandle node;
ros::Subscriber subInfo = node.subscribe("/vrep/info",1,infoCallback);
ros::ServiceClient client_enableSubscriber = node.serviceClient<vrep_common::simRosEnableSubscriber>("/vrep/simRosEnableSubscriber");

vrep_common::simRosEnableSubscriber srv_enableSubscriber;

srv_enableSubscriber.request.topicName = "/Position";
srv_enableSubscriber.request.streamCmd = simros_strmcmd_set_object_position;
srv_enableSubscriber.request.auxInt1 = handleOfTheObject;
srv_enableSubscriber.request.auxInt2 = -1;
srv_enableSubscriber.request.auxString = "";

if (client_enableSubscriber.call(srv_enableSubscriber) && (srv_enableSubscriber.responsesubscriberID != -1)) {
    ros::Publisher pubPosizione = node.advertise<geometry_msgs::Point>("Position",1);
    msg.x = -1;
    msg.y = 0;
    msg.z = 0.25;
    printf("conex. works!\n");

    while (ros::ok() && simulationRunning) {
        pubPosizione.publish(msg);
        ros::spinOnce();
        usleep(5000);
    }
    printf("simTerminated\n");
    return 0;
}

Now, the important step is, once the program is compiled, to copy the executable, located in /tesi/bin, in V-REP installation folder. In this way, it is not necessary to separately run the V-REP simulation and the ROS node as we did in the previous examples, but it is sufficient to start the simulation and the non-threaded child script will call the executable using the command simLunchExecutable.
Appendix C

Other

C.1 Proximity Sensors Calibration

\[ AS_1 = \begin{pmatrix} 0.0000 & 1.0000 & 0.0000 & 0.2400 \\ -1.0000 & 0.0000 & -0.0000 & -0.0000 \\ -0.0000 & -0.0000 & 1.0000 & 0.7890 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  \hspace{1cm} (C.1)

\[ AS_2 = \begin{pmatrix} -0.0000 & 1.0000 & 0.0000 & 0.2400 \\ -0.6235 & 0.0000 & -0.7818 & -0.0602 \\ -0.7818 & -0.0000 & 0.6235 & 0.7600 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  \hspace{1cm} (C.2)

\[ AS_3 = \begin{pmatrix} -0.0000 & 1.0000 & 0.0000 & 0.2400 \\ -0.6235 & 0.0000 & -0.7818 & -0.0602 \\ -0.7818 & -0.0000 & 0.6235 & 0.7600 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  \hspace{1cm} (C.3)

\[ AS_4 = \begin{pmatrix} -0.0000 & 1.0000 & -0.0000 & 0.2600 \\ 0.9010 & 0.0000 & -0.4339 & -0.0334 \\ -0.4339 & -0.0000 & -0.9010 & 0.6426 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  \hspace{1cm} (C.4)

\[ AS_5 = \begin{pmatrix} -0.0000 & 1.0000 & -0.0000 & 0.2400 \\ 0.9010 & 0.0000 & 0.4339 & 0.0334 \\ 0.4339 & -0.0000 & -0.9010 & 0.6426 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  \hspace{1cm} (C.5)
\[ AS_6 = \begin{pmatrix} 0.0000 & 1.0000 & -0.0000 & 0.2400 \\ 0.2225 & 0.0000 & 0.9749 & 0.0751 \\ 0.9749 & -0.0000 & -0.2225 & 0.6949 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  
(C.6)

\[ AS_7 = \begin{pmatrix} 0.0000 & 1.0000 & -0.0000 & 0.2400 \\ -0.6235 & 0.0000 & 0.7818 & 0.0602 \\ 0.7818 & -0.0000 & 0.6235 & 0.7600 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  
(C.7)

\[ AS_8 = \begin{pmatrix} 0 & 1.0000 & 0.0000 & 0.3850 \\ -1.0000 & 0 & -0.0000 & -0.0670 \\ -0.0000 & -0.0000 & 1.0000 & 0.7940 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  
(C.8)

\[ AS_9 = \begin{pmatrix} 0.0000 & 1.0000 & 0 & 0.4500 \\ -0.0000 & 0 & 1.0000 & 0.0930 \\ 1.0000 & -0.0000 & 0.0000 & 0.7120 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  
(C.9)

\[ AS_{17} = \begin{pmatrix} 0.0000 & 1.0000 & -0.0000 & 0.0700 \\ -0.0000 & 0.0000 & 1.0000 & 0.0950 \\ 1.0000 & -0.0000 & 0.0000 & 0.7120 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  
(C.10)

\[ AS_{20} = \begin{pmatrix} 0.0000 & -1.0000 & 0 & 0.4500 \\ -0.0000 & 0 & -1.0000 & -0.1650 \\ 1.0000 & 0.0000 & -0.0000 & 0.7120 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix} \]  
(C.11)
### C.2 Built-in standard ROS message types.

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>Serialization</th>
<th>C++</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Unsigned 8-bit int</td>
<td>uint8_t</td>
<td>bool</td>
</tr>
<tr>
<td>int8</td>
<td>Signed 8-bit int</td>
<td>int8_t</td>
<td>int</td>
</tr>
<tr>
<td>uint8</td>
<td>Unsigned 8-bit int</td>
<td>uint8_t</td>
<td>int</td>
</tr>
<tr>
<td>int16</td>
<td>Signed 16-bit int</td>
<td>int16_t</td>
<td>int</td>
</tr>
<tr>
<td>uint16</td>
<td>Unsigned 16-bit int</td>
<td>uint16_t</td>
<td>int</td>
</tr>
<tr>
<td>int32</td>
<td>Signed 32-bit int</td>
<td>int32_t</td>
<td>int</td>
</tr>
<tr>
<td>uint32</td>
<td>Unsigned 32-bit int</td>
<td>uint32_t</td>
<td>int</td>
</tr>
<tr>
<td>int64</td>
<td>Signed 64-bit int</td>
<td>int64_t</td>
<td>long</td>
</tr>
<tr>
<td>uint64</td>
<td>Unsigned 64-bit int</td>
<td>uint64_t</td>
<td>long</td>
</tr>
<tr>
<td>float32</td>
<td>32-bit IEEE float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>float64</td>
<td>64-bit IEEE float</td>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td>string</td>
<td>ASCII string (4-bit)</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>time</td>
<td>Secs/nsecs signed 32-bit ints</td>
<td>ros::Time</td>
<td>rospys.Time</td>
</tr>
<tr>
<td>duration</td>
<td>Secs/nsecs signed 32-bit ints</td>
<td>ros::Duration</td>
<td>rospys.Duration</td>
</tr>
</tbody>
</table>

*Table C.1: Built-in standard ROS message types.*
Appendix D

Nodes

D.1 Sensors

D.1.1 Proximity sensor (proxSensors.cpp)

```c++
// proximity sensors
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include "sensor_msgs/Image.h"
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"

#include "tesi/vett_proxData.h"
#include "tesi/proxData.h"
#include "vrep_common/simRosGetObjectHandle.h"
#include "vrep_common/simRosGetObjectGroupData.h"

ros::ServiceClient client_groupData;
vrep_common::simRosGetObjectGroupData srv_groupData;
tesi::proxData proxData;
tesi::vett_proxData vett_proxData;
ros::Publisher Prox_pub;
std::string nomeProx[11];

bool simulationRunning=true;
float simulationTime=0.0f;
void infoCallback(const vrep_common::VrepInfo::ConstPtr& info)
{
  simulationTime=info->simulationTime.data;
simulationRunning=(info->simulatorState.data&1)!=0;
}

void LeggiGroupData()
{
  vett_proxData.proxData.data.clear();
srv_groupData.request.dataType=13; // proximity sensor infos
  if (client_groupData.call(srv_groupData))
  {
```
ROS_INFO("\nInizio pubblicazione informazioni proximity sensors\n");
for(int i=0; i<11; i++)
{
    if(srv_groupData.response.intData[i*2]==0)
    {
        proxData.state=0;
        proxData.objectHandle=0;
        proxData.point[0]=0;
        proxData.point[1]=0;
        proxData.point[2]=0;
        proxData.normal[0]=0;
        proxData.normal[1]=0;
        proxData.normal[2]=0;
    }
    else
    {
        // in intData
        proxData.state=srv_groupData.response.intData[i*2];
        proxData.objectHandle=srv_groupData.response.intData[i*2+1];
        // in floatData
        proxData.point[0]=srv_groupData.response.floatData[i*6];
        proxData.point[1]=srv_groupData.response.floatData[i*6+1];
        proxData.point[2]=srv_groupData.response.floatData[i*6+2];
        proxData.normal[0]=srv_groupData.response.floatData[i*6+3];
        proxData.normal[1]=srv_groupData.response.floatData[i*6+4];
        proxData.normal[2]=srv_groupData.response.floatData[i*6+5];
    }
    proxData.nomeProx=nomeProx[i];
    vett_proxData.proxData.push_back(proxData);
}
Prox_pub.publish(vett_proxData);

int main(int argc, char **argv)
{
    ros::init(argc, argv, "Proximity");
    ros::NodeHandle node;
    ros::Subscriber subInfo=node.subscribe("/vrep/info",1,infoCallback);
    client_groupData=node.serviceClient<vrep_common::simRosGetObjectGroupData>("/vrep/simRosGetObjectGroupData");
    srv_groupData.request.objectType=sim_object_proximitysensor_type;
    srv_groupData.request.dataType=0;
    if(client_groupData.call(srv_groupData))
    {
        for(int j=0; j<11; j++)
            nomeProx[j]=srv_groupData.response.strings[j];
    }
    Prox_pub = node.advertise<tesi::vett_proxData>("prox_data", 1000);
    while((ros::ok())&&simulationRunning)
    {
        LeggiGroupData();
        ros::spinOnce();
        usleep(5000);
    }
    LOG_INFO("\nSimulation ended! \n");
    return 0;
}
D.1.2 Kinect (kinect.cpp)

```c
//kinect
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/rost.h"
#include <iostream>

#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"

#include "vrep_common/simRosGetObjectHandle.h"
#include "vrep_common/simRosGetVisionSensorDepthBuffer.h"

ros::ServiceClient client_getVisionSensorDepthBuffer;

vrep_common::simRosGetVisionSensorDepthBuffer srv_getVisionSensorDepthBuffer;
bool simulationRunning=true;
float simulationTime=0.0f;

void infoCallback(const vrep_common::VrepInfo::ConstPtr & info)
{
    simulationTime=info->simulationTime.data;
    simulationRunning=(info->simulatorState.data&1)!=0;
}

void LeggiDepth()
{
    int j=0;
    ROS_INFO("Stampo buffer:\n");
    for(int i=0;i<\(\*srv_getVisionSensorDepthBuffer.response.resolution0*\)
\(\*srv_getVisionSensorDepthBuffer.response.resolution1\));i++)
    {
        j++;
        if(j==64)
        {
            printf("\n");
            j=0;
        }
        printf("[\%d]\", i);
        printf("%.3f - ", srv_getVisionSensorDepthBuffer.response.buffer[i]);
    }
}

int main(int argc, char **argv)
{
    int kinectHandle;
    
    ros::init(argc, argv, "Kinect");
    ros::NodeHandle node;
    ros::Subscriber subInfo=node.subscribe("/vrep/info",1,infoCallback);

    ros::ServiceClient client_getObjectHandle=node.serviceClient<vrep_common::
simRosGetObjectHandle>("/vrep/simRosGetObjectHandle");
    vrep_common::simRosGetObjectHandle srv_getObjectHandle;
    if(client_getObjectHandle.call(srv_getObjectHandle))
        kinectHandle=srv_getObjectHandle.response.handle;
    client_getVisionSensorDepthBuffer=node.serviceClient<vrep_common::
simRosGetVisionSensorDepthBuffer>("/vrep/simRosGetVisionSensorDepthBuffer");
    srv_getVisionSensorDepthBuffer.request.handle=kinectHandle;
    if((client_getVisionSensorDepthBuffer.call(srv_getVisionSensorDepthBuffer))
        &&(srv_getVisionSensorDepthBuffer.response.result!=1))
    {
        ROS_INFO("\nInizio lettura depth sensor \n");
        ROS_INFO("\nRisoluzione %d x %d", srv_getVisionSensorDepthBuffer.response
.resolution0,srv_getVisionSensorDepthBuffer.response.resolution1);
        while (ros::ok()&&simulationRunning)
        {
            LeggiDepth();
        }
    }
    
    return 0;
}
```

```cpp
ros::spinOnce();
usleep(5000);

ROS_INFO("\nSimulation ended! \n");
return 0;
```
# D.1.3 Node telecamera_0 (telecamera_0.cpp)

```c++
// telecamera_0
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include <sensor_msgs/Image.h>
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"

ros::ServiceClient client_getVisionSensorImage;
ros::Publisher Camera_Image_pub;

bool simulationRunning = true;
float simulationTime = 0.0f;

void infoCallback(const vrep_common::VrepInfo::ConstPtr & info)
{
    simulationTime = info->simulationTime.data;
    simulationRunning = (info->simulatorState.data &1) != 0;
}

void LeggiImmagine()
{
    client_getVisionSensorImage.call(srv_getVisionSensorImage);
    Camera_Image_pub.publish(srv_getVisionSensorImage.response.image);
}

int main(int argc, char **argv)
{
    int cameraHandle;
    ros::init(argc, argv, "telecamera_0");
    ros::NodeHandle node;
    ros::Subscriber subInfo = node.subscribe("/vrep/info", 1, infoCallback);
    ros::ServiceClient client_getObjectHandle = node.serviceClient<vrep_common::simRosGetObjectHandle>("/vrep/simRosGetObjectHandle");
    vrep_common::simRosGetObjectHandle srv_getObjectHandle;
    srv_getObjectHandle.request.objectName = "camera0";
    if (client_getObjectHandle.call(srv_getObjectHandle))
        cameraHandle = srv_getObjectHandle.response.handle;
    Camera_Image_pub = node.advertise<sensor_msgs::Image>("Camera_Image_0", 1000);
    srv_getVisionSensorImage.request.handle = cameraHandle;
    srv_getVisionSensorImage.request.options = 0;
    if (((client_getVisionSensorImage.call(srv_getVisionSensorImage)) &&
        srv_getVisionSensorImage.response.result != -1))
    {
        ROS_INFO("Inizio lettura vision sensor -camera0-\n");
        while (ros::ok() && simulationRunning)
        {
            LeggiImmagine();
            ros::spinOnce();
            usleep(5000);
        }
    }
    ROS_INFO("InSimulation ended! \n");
```
66  return 0;
67  }

D.1.4 Node telecamera_1 (telecamera_1.cpp)

```c++
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include "sensor_msgs/Image.h"
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"
#include "vrep_common/simRosGetObjectHandle.h"
#include "vrep_common/simRosGetVisionSensorImage.h"

ros::ServiceClient client_getVisionSensorImage;
vrep_common::simRosGetVisionSensorImage srv_getVisionSensorImage;
ros::Publisher Camera_Image_pub;

bool simulationRunning=true;0
float simulationTime=0.0f;
void infoCallback(const vrep_common::VrepInfo::ConstPtr & info)
{
    simulationTime=info->simulationTime.data;
    simulationRunning=(info->simulatorState.data&1)!=0;
}

void LeggiImmagine()
{
    client_getVisionSensorImage.call(srv_getVisionSensorImage);
    Camera_Image_pub.publish(srv_getVisionSensorImage.response.image);
}

int main(int argc, char **argv)
{
    int cameraHandle;
    ros::init(argc, argv, "telecamera_1");
    ros::NodeHandle node;
    ros::Subscriber subInfo=node.subscribe("/vrep/info",1,infoCallback);
    ros::ServiceClient client_getObjectHandle=node.serviceClient<vrep_common::simRosGetObjectHandle>("/vrep/simRosGetObjectHandle");
vrep_common::simRosGetObjectHandle srv_getObjectHandle;
    srv_getObjectHandle.request.objectName="camera";
    if(client_getObjectHandle.call(srv_getObjectHandle))
        cameraHandle=srv_getObjectHandle.response.handle;
    Camera_Image_pub = node.advertise<sensor_msgs::Image>("Camera_Image_1", 1000);
    srv_getVisionSensorImage.request.handle=cameraHandle;
    srv_getVisionSensorImage.request.options=0;
    if((client_getVisionSensorImage.call(srv_getVisionSensorImage))&&
        (srv_getVisionSensorImage.response.result!=-1))
    {
        ROS_INFO("\nReading vision sensor -camera1- \n");
        while (ros::ok()&&simulationRunning)
        {
            LeggiImmagine();
            ros::spinOnce();
            usleep(5000);
        }
    }
    ROS_INFO("\nSimulation ended! \n");
```
return 0;
D.2 Robots

D.2.1 robot (robot.cpp)

```c++
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include "tesi/ComauMsg.h"
#include "tesi/IrbMsg.h"
#include "tesi/YouBotMsg.h"
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"
#include "vrep_common/simRosGetObjectHandle.h"

bool simulationRunning = true;
float simulationTime = 0.0f;

void infoCallback(const vrep_common::VrepInfo::ConstPtr & info) {
  simulationTime = info->simulationTime.data;
  simulationRunning = (info->simulatorState.data & 1) != 0;
}

int main(int argc, char **argv) {
  tesi::ComauMsg ComauMsg;
  tesi::IrbMsg IrbMsg;
  tesi::YouBotMsg YouBotMsg;
  int argc = 33;
  if (argc > 1) {
    ComauMsg.hComauRobot = atoi(argv[1]);
    IrbMsg.hIrbRobot = atoi(argv[2]);
    YouBotMsg.hYouBotRobot = atoi(argv[3]);
    for (int j = 0; j < 6; j++)
      IrbMsg.hIrbJoint[j] = atoi(argv[4 + j]);
    for (int i = 0; i < 7; i++)
      ComauMsg.hComauJoint[i] = atoi(argv[10 + i]);
    for (int k = 0; k < 16; k++)
      YouBotMsg.hYouBotJoint[k] = atoi(argv[17 + k]);
  } else {
    printf("Indicate following arguments: 'handle COMAU, handle IRB e handle YOUBOT '!\n");
    sleep(5000);
    return 0;
  }

  int argc = 0;
  char ** argv = NULL;
  ros::init(argc, argv, "robot");
  if (ros::master::check())
    return 0;

  ros::NodeHandle node;
  ros::Subscriber subInfo = node.subscribe="/vrep/info", 1, infoCallback);
  printf("nCOMAU: %d, IRB: %d, YOUBOT: %d\n", ComauMsg.hComauRobot, IrbMsg.
    hIrbRobot, YouBotMsg.hYouBotRobot);
  printf("nCOMAU: %d, IRB: %d, YOUBOT: %d\n", ComauMsg.hComauJoint[1]);
  for (int j = 0; j < 6; j++)
    printf("%d, ", IrbMsg.hIrbJoint[j]);
  for (int i = 0; i < 7; i++)
    printf("%d, ", ComauMsg.hComauJoint[i]);

```
printf("\n\nYOUBOT: ");
for(int k=0;k<16;k++)
  printf("%d, ",YouBotMsg.hYouBotJoint[k]);
printf("\n");
ros::Publisher Comau_pub = node.advertise<tesi::ComauMsg>("ComauHandles", 1000);
ros::Publisher Irb_pub = node.advertise<tesi::IrbMsg>("IrbHandles", 1000);
ros::Publisher YouBot_pub = node.advertise<tesi::YouBotMsg>("YouBotHandles", 1000);
while (ros::ok()&&simulationRunning)
{
  Comau_pub.publish(ComauMsg);
  Irb_pub.publish(IrbMsg);
  YouBot_pub.publish(YouBotMsg);
  ros::spinOnce();
  usleep(5000);
}
printf("\nSimulation ended!\n");
return 0;
D.2.2 YouBotHandles (YouBot.cpp)

```c++
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include "sensor_msgs/JointState.h"
#include "std_msgs/Header.h"
#include "tesi/YouBotMsg.h"
#include "tesi/JointHandles.h"
#include "tesi/vett_JointHandles.h"
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"
#include "vrep_common/simRosGetObjectHandle.h"
#include "vrep_common/simRosGetObjectGroupData.h"

tesi::YouBotMsg YouBotMsg;
tesi::JointHandles jointHandles;
tesi::vett_JointHandles vett_jointHandles;
ros::ServiceClient client_groupData;
vrep_common::simRosGetObjectGroupData srv_groupData;
ros::Publisher YouBot_JointHandles_pub;

bool simulationRunning = true;
float simulationTime = 0.0f;

void infoCallback(const vrep_common::VrepInfo::ConstPtr & info) {
    simulationTime = info->simulationTime.data;
    simulationRunning = (info->simulatorState.data & 1) != 0;
}

void chatterCallback(const tesi::YouBotMsg::ConstPtr & Msg) {
    YouBotMsg = Msg;
}

void LeggiGroupData() {
    vett_jointHandles.handles.clear();
    srv_groupData.request.dataType = 0;
    bool trigger = false;
    if (client_groupData.call(srv_groupData)) {
        for (int k = 0; k < 16; k++)
            for (int i = 0; i < 30; i++)
                if (YouBotMsg.hYouBotJoint[k] != 0 && (srv_groupData.response.handles[i] == YouBotMsg.hYouBotJoint[k])) {
                    jointHandles.indice = i;
                    jointHandles.handle = srv_groupData.response.handles[i];
                    jointHandles.nome = srv_groupData.response.strings[i];
                    vett_jointHandles.handles.push_back(jointHandles);
                    trigger = true;
                }
    } else {
        return;
    }
    if (trigger == true)
        YouBot_JointHandles_pub.publish(vett_jointHandles);
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "YouBot");
    return 0;
}
```

D.2. ROBOTS

99
roscpp::NodeHandle node;
ros::Subscriber subInfo = node.subscribe("/vrep/info",1,infoCallback);
ros::Subscriber sub = node.subscribe("YouBotHandles", 1000, chatterCallback);
std::string client_groupData = node.serviceClient<vrep_common::simRosGetObjectGroupData>("/vrep/simRosGetObjectGroupData");
srv_groupData.request.objectType = sim_object_joint_type;
YouBot_JointHandles_pub = node.advertise<tesi::vett_JointHandles>("YouBot_JointHandles", 1000);
while (ros::ok()&&simulationRunning)
{
    LeggiGroupData();
    ros::spinOnce();
    usleep(5000);
}
ROS_INFO("\nSimulation ended! \n");
return 0;
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/rosc.h"
#include <iostream>
#include <sensor_msgs/JointState.h>
#include <std_msgs/Header.h>
#include "tesi/IrbMsg.h"
#include <tesi/JointHandles.h>
#include "tesi/vett_JointHandles.h"
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"
#include "vrep_common/simRosGetObjectHandle.h"
#include "vrep_common/simRosGetObjectGroupData.h"

// -----------------
#include "vrep_common/simRosGetObjectGroupData.h"
tesi::IrbMsg IrbMsg;
tesi::JointHandles jointHandles;
tesi::vett_JointHandles vett_jointHandles;
ros::ServiceClient client_groupData;
vrep_common::simRosGetObjectGroupData srv_groupData;
ros::Publisher Irb_JointHandles_pub;

bool simulationRunning=true;
float simulationTime=0.0f;

void infoCallback(const vrep_common::VrepInfo::ConstPtr& info)
{
  simulationTime=info->simulationTime.data;
  simulationRunning=(info->simulatorState.data&1)!=0;
}

void chatterCallback(const tesi::IrbMsg::ConstPtr& Msg)
{
  IrbMsg=Msg;
}

void LeggiGroupData()
{
  vett_jointHandles.handles.clear();
srv_groupData.request.dataType=0;
bool trigger=false;
if(client_groupData.call(srv_groupData))
{
  for(int k=0;k<6;k++)
    for(int i=0;i<30;i++)
      if((IrbMsg.hIrbJoint[k]!=0)&&(srv_groupData.response.handles[i]==IrbMsg.hIrbJoint[k]))
      {
        jointHandles.indice = i;
jointHandles.handle=srv_groupData.response.handles[i];
jointHandles.nome=srv_groupData.response.strings[i];
vett_jointHandles.handles.push_back(jointHandles);
      trigger=true;
      }
  }
  else
    return;
if(trigger==true)
  Irb_JointHandles_pub.publish(vett_jointHandles);
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "Irb");
```cpp
ros::NodeHandle node;
ros::Subscriber subInfo = node.subscribe("/vrep/info", 1, infoCallback);
ros::Subscriber sub = node.subscribe("IrbHandles", 1000, chatterCallback);

client_groupData = node.serviceClient<vrep_common::simRosGetObjectGroupData>(
    "/vrep/simRosGetObjectGroupData");
srv_groupData.request.objectType = sim_object_joint_ty pe;
Irb_JointHandles_pub = node.advertise<tesi::vett_JointHandles>(
    "Irb_JointHandles", 1000);

while (ros::ok() && simulationRunning) {
    LeggiGroupData();
    ros::spinOnce();
    usleep(5000);
}
ROS_INFO("\nSimulation ended! \n");
return 0;
}```
### D.2.4 ComauHandles (Comau.cpp)

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include <sensor_msgs/JointState.h>
#include <std_msgs/Header.h>
#include "tesi/ComauMsgr.h"
#include <tesi/JointHandles.h>
#include <tesi/vett_JointHandles.h>
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"
#include "vrep_common/simRosGetObjectHandle.h"
#include "vrep_common/simRosGetObjectGroupData.h"
tesi::ComauMsg ComauMsg;
tesi::JointHandles jointHandles;
tesi::vett_JointHandles vett_jointHandles;
ros::ServiceClient client_groupData;
vrep_common::simRosGetObjectGroupData srv_groupData;
ros::Publisher Comau_JointHandles_pub;

bool simulationRunning = true;
float simulationTime = 0.0f;
void infoCallback(const vrep_common::VrepInfo::ConstPtr & info)
{
    simulationTime = info->simulationTime.data;
    simulationRunning = (info->simulatorState.data & 1) != 0;
}

void chatterCallback(const tesi::ComauMsg::ConstPtr & Msg)
{
    ComauMsg = *Msg;
}

void LeggiGroupData()
{
    vett_jointHandles.handles.clear();
    srv_groupData.request.dataType = 0;
    bool trigger = false;
    if (client_groupData.call(srv_groupData))
    {
        for(int k = 0; k < 7; k++)
            for(int i = 0; i < 30; i++)
                if((ComauMsg.hComauJoint[k] != 0) && (srv_groupData.response.handles[i] == ComauMsg.hComauJoint[k]))
                {
                    jointHandles.indice = i;
                    jointHandles.handle = srv_groupData.response.handles[i];
                    jointHandles.nome = srv_groupData.response.strings[i];
                    vett_jointHandles.handles.push_back(jointHandles);
                    trigger = true;
                }
    } else
    return;

    if (trigger == true)
    Comau_JointHandles_pub.publish(vett_jointHandles);
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "Comau");
    return 0;
}
```
ros::NodeHandle node;
ros::Subscriber subInfo = node.subscribe("/vrep/info",1,infoCallback);
ros::Subscriber sub = node.subscribe("ComauHandles", 1000, chatterCallback);

client_groupData = node.serviceClient<vrep_common::simRosGetObjectGroupData>(
   "/vrep/simRosGetObjectGroupData");
srv_groupData.request.objectType = sim_object_joint_type;
Comau_JointHandles_pub = node.advertise<tesi::vett_JointHandles>("Comau_JointHandles", 1000);

while (ros::ok() && simulationRunning)
{
   LeggiGroupData();
   ros::spinOnce();
   usleep(5000);
}
ROS_INFO("\nSimulation ended! \n");
return 0;
2.5 NYouBot_PosOrRel (NYouBot_PosOrRel.cpp)

```c++
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include "tesi/YouBotMsg.h"
#include "sensor_msgs/JointState.h"
#include <std_msgs/Header.h>
// nuovi messaggi
#include "tesi/vett_posOrRelGiunto.h"
#include "tesi/posOrRelGiunto.h"
#include "tesi/JointHandles.h"
#include "tesi/vett_JointHandles.h"

// -----------------

tesi::vett_posOrRelGiunto vett_posOrRelGiunto;
tesi::posOrRelGiunto posOrRelGiunto;
tesi::vett_JointHandles vett_JointHandles;
// -----------------

#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"

#include "vrep_common/simRosGetJointState.h"
#include "vrep_common/simRosGetObjectGroupData.h"

tesi::YouBotMsg YouBotMsg;
ros::Publisher YouBotPosOrRel_pub;
bool simulationRunning=true;
float simulationTime=0.0f;
void infoCallback(const vrep_common::VrepInfo::ConstPtr & info)
{
    simulationTime=info->simulationTime.data;
    simulationRunning=(info->simulatorState.data&1)!=0;
}

bool trigger=false;
void chatterCallback(const tesi::vett_JointHandles::ConstPtr & Msg)
{
    vett_JointHandles(handles).clear();
    vett_JointHandles=**Msg;
    trigger=true;
}

ros::ServiceClient client_groupData;
vrep_common::simRosGetObjectGroupData srv_groupData;
void LeggiGroupData()
{
    vett_posOrRelGiunto.posizioneOrientamento.clear();
srv_groupData.request.dataType=9;
if(client_groupData.call(srv_groupData))
{
    for(int k=0;k<16;k++)
    {
        posOrRelGiunto.Posizione_Relativa[0]=srv_groupData.response.floatData[(
vett_JointHandles(handles)[k].indice)*6];
        posOrRelGiunto.Posizione_Relativa[1]=srv_groupData.response.floatData[(
vett_JointHandles(handles)[k].indice)*6+1];
        posOrRelGiunto.Orientamento_Relativo[0]=srv_groupData.response.floatData[(
vett_JointHandles(handles)[k].indice)*6+2];
        posOrRelGiunto.Orientamento_Relativo[1]=srv_groupData.response.floatData[(
vett_JointHandles(handles)[k].indice)*6+3];
        posOrRelGiunto.Orientamento_Relativo[2]=srv_groupData.response.floatData[(
vett_JointHandles(handles)[k].indice)*6+4];
    }
}
```
posOrRelGiunto.Orientamento_Relativo[2]=srv_groupData.response.floatData[(vett_JointHandles(handles[k].indice)*6+5);
posOrRelGiunto.headerGiunto.handle=vett_JointHandles(handles[k].handle;
posOrRelGiunto.headerGiunto.nome=vett_JointHandles(handles[k].nome;
vett_posOrRelGiunto.posizioneOrientamento.push_back(posOrRelGiunto);}
else
{ROS_WARN("\n Connecting...!\n");return;
}
YouBotPosOrRel_pub.publish(vett_posOrRelGiunto);
}

int main(int argc, char **argv)
{
ros::init(argc, argv, "NYouBot_PosOrRel");
ros::NodeHandle node;
ros::Subscriber subInfo=node.subscribe("/vrep/info",1,infoCallback);
ros::Subscriber sub = node.subscribe("YouBot_JointHandles", 1000, chatterCallback);
client_groupData=node.serviceClient<vrep_common::simRosGetObjectGroupData>("/vrep/simRosGetObjectGroupData");
srv_groupData.request.objectType=sim_object_joint_type;
YouBotPosOrRel_pub = node.advertise<tesi::vett_posOrRelGiunto>("YouBot_PosOrRel", 1000);
while (ros::ok() && simulationRunning)
{
ros::spinOnce();
if(trigger==true)
  LeggiGroupData();
usleep(5000);
ROS_INFO("\nSimulation ended! \n");return 0;
}
D.2.6 NYYouBot_PosOrAss (NYYouBot_PosOrAss.cpp)

```c++
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include "tesi/YouBotMsg.h"
#include "tesi/jointData.h"
#include <sensor_msgs/JointState.h>
#include <std_msgs/Header.h>
// new messages
#include "tesi/vett_posOrAssGiunto.h"
#include "tesi/posOrAssGiunto.h"
#include <tesi/JointHandles.h>
#include <tesi/vett_JointHandles.h>

// -----------------

tesi::vett_JointHandles vett_JointHandles;

// -----------------

#include "vrep_common/VrepInfo.h"
#include "vrep_common/vrep_plugin/include/v_repConst.h"

// API usati
#include "vrep_common/simRosGetJointState.h"
#include "vrep_common/simRosGetObjectGroupData.h"

tesi::YouBotMsg YouBotMsg;
ros::Publisher YouBotPosOrAss_pub;

bool simulationRunning=true;
float simulationTime=0.0f;

void infoCallback(const vrep_common::VrepInfo::ConstPtr & info)
{
  simulationTime=info->simulationTime.data;
  simulationRunning=(info->simulatorState.data&1)!=0;
}

bool trigger=false;

void chatterCallback(const tesi::vett_JointHandles::ConstPtr & Msg)
{
  vett_JointHandles handles.clear();
  vett_JointHandles=**Msg;
  trigger=true;
}

ros::ServiceClient client_groupData;
ros::ServiceClient client_groupData;
void LeggiGroupData()
{
  vett_posOrAssGiunto posizioneOrientamento.clear();
  srv_groupData.request.dataType=9;
  if(client_groupData.call(srv_groupData))
  {
    for(int k=0;k<16;k++)
    {
      posOrAssGiunto.Posizione_Assoluto[0]=srv_groupData.response.floatData[(
        vett_JointHandles handles[k].indice)*6];
      posOrAssGiunto.Posizione_Assoluto[1]=srv_groupData.response.floatData[(
        vett_JointHandles handles[k].indice)*6+1];
      posOrAssGiunto.Posizione_Assoluto[2]=srv_groupData.response.floatData[(
        vett_JointHandles handles[k].indice)*6+2];
      posOrAssGiunto.Orientamento_Assoluto[0]=srv_groupData.response.floatData[(
        vett_JointHandles handles[k].indice)*6+3];
      posOrAssGiunto.Orientamento_Assoluto[1]=srv_groupData.response.floatData[(
        vett_JointHandles handles[k].indice)*6+4];
    }
  }
}
```
posOrAssGiunto.Orientamento_Assoluto[2]=srv_groupData.response.floatData
[(vett_JointHandles.handles[k].indice)*6+5];
posOrAssGiunto.headerGiunto.handle=vett_JointHandles.handles[k].handle;
posOrAssGiunto.headerGiunto.nome=vett_JointHandles.handles[k].nome;
vett_posOrAssGiunto.posizioneOrientamento.push_back(posOrAssGiunto);
}
} else
{
  ROS_WARN("\n Connecting...!\n");
  return;
}
YouBotPosOrAss_pub.publish(vett_posOrAssGiunto);
}
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "NYouBot_PosOrAss");
  ros::NodeHandle node;
  ros::Subscriber subInfo=node.subscribe("/vrep/info",1,infoCallback);
  ros::Subscriber sub = node.subscribe("YouBot_JointHandles", 1000,
    chatterCallback);
  client_groupData = node.serviceClient<vrep_common::simRosGetObjectGroupData>("/
vrep/simRosGetObjectGroupData");
  srv_groupData.request.objectType=sim_object_joint_type;
  YouBotPosOrAss_pub = node.advertise<tesi::vett_posOrAssGiunto>("YouBot_PosOrAss", 1000);
  while (ros::ok()&&simulationRunning)
  {
    ros::spinOnce();
    if(trigger==true)
      LeggiGroupData();
    usleep(5000);
  }
  ROS_INFO("\nSimulation ended! \n");
  return 0;
}
D.2.7 NIrb_PosOrRel (NIrb_PosOrRel.cpp)

```c++
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include "tesi/IrbMsg.h"
#include "tesi/jointData.h"
#include "sensor_msgs/JointState.h"
#include "std_msgs/Header.h"

// new messages
#include "tesi/vett_posOrRelGiunto.h"
#include "tesi/posOrRelGiunto.h"
#include "tesi/JointHandles.h"
#include "tesi/vett_JointHandles.h"

// -----------------

tesi::vett_posOrRelGiunto vett_posOrRelGiunto ;
tesi::posOrRelGiunto posOrRelGiunto ;
tesi::vett_JointHandles vett_JointHandles ;
// -----------------

#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"

#include "vrep_common/simRosGetJointState.h"
#include "vrep_common/simRosGetObjectGroupData.h"

tesi::IrbMsg IrbMsg ;
ros::Publisher IrbPosOrRel_pub ;

bool simulationRunning =true ;
float simulationTime =0.0f ;

void infoCallback (const vrep_common::VrepInfo::ConstPtr & info )
{
    simulationTime =info ->simulationTime . data ;
    simulationRunning =( info ->simulatorState.data &1) !=0 ;
}

bool trigger =false ;

void chatterCallback (const tesi::vett_JointHandles::ConstPtr & Msg )
{
    vett_JointHandles . handles . clear () ;
    vett_JointHandles =* Msg ;
    trigger =true ;
}

ros::ServiceClient client_groupData ;
vrep_common::simRosGetObjectGroupData srv_groupData ;

void LeggiGroupData ()
{
    vett_posOrRelGiunto . posizioneOrientamento . clear () ;
    srv_groupData . request . dataType =10 ;
    if (client_groupData . call (srv_groupData ))
    {
        for (int k =0 ; k <6 ; k++)
        {
            posOrRelGiunto . Posizione_Relativa [0] =srv_groupData . response . floatData [(
                vett_JointHandles . handles [k] . indice )*6 ] ;
            posOrRelGiunto . Posizione_Relativa [1] =srv_groupData . response . floatData [(
                vett_JointHandles . handles [k] . indice )*6+1 ] ;
            posOrRelGiunto . Posizione_Relativa [2] =srv_groupData . response . floatData [(
                vett_JointHandles . handles [k] . indice )*6+2 ] ;
            posOrRelGiunto . Orientamento_Relativo [0] =srv_groupData . response . floatData [
                (vett_JointHandles . handles [k] . indice )*6+3 ] ;
            posOrRelGiunto . Orientamento_Relativo [1] =srv_groupData . response . floatData [
                (vett_JointHandles . handles [k] . indice )*6+4 ] ;
        }
    }
}```
posOrRelGiunto.Orientamento_Relativo[2]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6+5];
posOrRelGiunto.headerGiunto.handle=vett_JointHandles.handles[k].handle;
posOrRelGiunto.headerGiunto.nome=vett_JointHandles.handles[k].nome;
vett_posOrRelGiunto.posizioneOrientamento.push_back(posOrRelGiunto);
}
else
{
    ROS_WARN("\n Connecting...!\n");
    return;
}
IrbPosOrRel_pub.publish(vett_posOrRelGiunto);
}
int main(int argc, char **argv)
{
    ros::init(argc, argv, "NIrb_PosOrReL");
    ros::NodeHandle node;
    ros::Subscriber subInfo=node.subscribe("/vrep/info",1,infoCallback);
    ros::Subscriber sub = node.subscribe("Irb_JointHandles", 1000, chatterCallback);
    client_groupData=node.serviceClient<vrep_common::simRosGetObjectGroupData>("/vrep/simRosGetObjectGroupData");
    srv_groupData.request.objectType=sim_object_joint_type;
    IrbPosOrRel_pub = node.advertise<tesi::vett_posOrRelGiunto>("Irb_PosOrRel", 1000);
    while (ros::ok()&&simulationRunning)
    {
        ros::spinOnce();
        if(trigger==true)
            LeggiGroupData();
        usleep(5000);
    }
    ROS_INFO("\nSimulation ended! \n");
    return 0;
}
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include "tesi/IrbMsg.h"
#include "tesi/jointData.h"
#include <sensor_msgs/JointState.h>
#include <std_msgs/Header.h>

// new messages
#include "tesi/vett_posOrAssGiunto.h"
#include "tesi/posOrAssGiunto.h"
#include "tesi/JointHandles.h"
#include "tesi/vett_JointHandles.h"

// -----------------
tesi::vett_posOrAssGiunto vett_posOrAssGiunto;
tesi::posOrAssGiunto posOrAssGiunto;
tesi::vett_JointHandles vett_JointHandles;

// -----------------
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"

// API usati
#include "vrep_common/vrepcommonGetJointState.h"
#include "vrep_common/vrepcommonGetObjectGroupData.h"
tesi::IrbMsg IrbMsg;
ros::Publisher IrbPosOrAss_pub;

bool simulationRunning=true;
float simulationTime=0.0f;
void infoCallback(const vrep_common::VrepInfo::ConstPtr & info)
{
    simulationTime=info->simulationTime.data;
    simulationRunning=(info->simulatorState.data&1)!=0;
}

bool trigger=false;
void chatterCallback(const tesi::vett_JointHandles::ConstPtr & Msg)
{
    vett_JointHandles.handles.clear();
    vett_JointHandles=**Msg;
    trigger=true;
}

ros::ServiceClient client_groupData;
void LeggiGroupData()
{
    vett_posOrAssGiunto.posizioneOrientamento.clear();
srv_groupData.request.dataType=9;
    if(client_groupData.call(srv_groupData))
    {
        for(int k=0;k<6;k++)
        {
            posOrAssGiunto.Posizione_Assoluta[0]=srv_groupData.response.floatData[(((vett_JointHandles.handles[k].indice)*6)];
            posOrAssGiunto.Posizione_Assoluta[1]=srv_groupData.response.floatData[(((vett_JointHandles.handles[k].indice)*6+1)];
            posOrAssGiunto.Posizione_Assoluta[2]=srv_groupData.response.floatData[(((vett_JointHandles.handles[k].indice)*6+2)];
            posOrAssGiunto.Orientamento_Assoluto[0]=srv_groupData.response.floatData[(((vett_JointHandles.handles[k].indice)*6+3)];
            posOrAssGiunto.Orientamento_Assoluto[1]=srv_groupData.response.floatData[(((vett_JointHandles.handles[k].indice)*6+4)];
        }
    }
}
posOrAssGiunto.Orientamento_Absoluto[2]=srv_groupData.response.floatData
[(vett_JointHandles(handles[k].indice)*6+5];
posOrAssGiunto.headerGiunto.handle=vett_JointHandleshandles[k].handle;
posOrAssGiunto.headerGiunto.nome=vett_JointHandleshandles[k].nome;
vett_posOrAssGiunto.posizioneOrientamento.push_back(posOrAssGiunto);
}
else
{
ROS_WARN("\n Connecting...!\n");
return;
}
IrbPosOrAss_pub.publish(vett_posOrAssGiunto);
}
int main(int argc, char **argv)
{
ros::init(argc, argv, "NIrb_PosOrAss");
ros::NodeHandle node;
ros::Subscriber subInfo=node.subscribe("/vrep/info",1,infoCallback);
ros::Subscriber sub=node.subscribe("Irb_JointHandles", 1000,
chatterCallback);
client_groupData=node.serviceClient<vrep_common::simRosGetObjectGroupData>("/
/vrep/simRosGetObjectGroupData");
srv_groupData.request.objectType=sim_object_joint_type;
IrbPosOrAss_pub = node.advertise<tesi::vett_posOrAssGiunto>("Irb_PosOrAss", 
1000);
while (ros::ok()&&simulationRunning)
{
ros::spinOnce();
if(trigger==true)
   LeggiGroupData();
usleep(5000);
}
ROS_INFO("\nSimulation ended! \n");
return 0;
}
D.2.9 NComau_PosOrRel (NComau_PosOrRel.cpp)

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include "tesi/ComauMsg.h"
#include "sensor_msgs/JointState.h"
#include <std_msgs/Header.h>

// new messages
#include "tesi/vett_posOrRelGiunto.h"
#include "tesi/posOrRelGiunto.h"
#include "tesi/JointHandles.h"
#include "tesi/vett_JointHandles.h"

// -----------------

tesi::vett_posOrRelGiunto vett_posOrRelGiunto;
tesi::posOrRelGiunto posOrRelGiunto;
tesi::vett_JointHandles vett_JointHandles;
// -----------------
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"

#include "vrep_common/simRosGetJointState.h"
#include "vrep_common/simRosGetObjectGroupData.h"
tesi::ComauMsg ComauMsg;
ros::Publisher ComauPosOrRel_pub;

bool simulationRunning=true;
float simulationTime=0.0f;
void infoCallback(const vrep_common::VrepInfo::ConstPtr & info)
{
    simulationTime=info->simulationTime.data;
    simulationRunning=(info->simulatorState.data&1)!=0;
}

bool trigger=false;
void chatterCallback(const tesi::vett_JointHandles::ConstPtr & Msg)
{
    vett_JointHandles.handles.clear();
    vett_JointHandles=**Msg;
    trigger=true;
}

ros::ServiceClient client_groupData;
ros::ServiceClient client_groupData_srv_groupData;

void LeggiGroupData()
{
    vett_posOrRelGiunto.posizioneOrientamento.clear();
srv_groupData.request.dataType=10;
if(client_groupData.call(srv_groupData))
{
    for(int k=0;k<7;k++)
    {
        posOrRelGiunto.Posizione_Relativa[0]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6];
        posOrRelGiunto.Posizione_Relativa[1]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6+1];
        posOrRelGiunto.Posizione_Relativa[2]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6+2];
        posOrRelGiunto.Orientamento_Relativo[0]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6+3];
        posOrRelGiunto.Orientamento_Relativo[1]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6+4];
    }
}
```
```cpp
posOrRelGiunto.Orientamento_Relativo[2]=srv_groupDa.ta .response.floatData
[(vett_JointHandles.handles[k].indice)*6+5];
posOrRelGiunto.headerGiunto.handle=vett_JointHandles.handles[k].handle;
posOrRelGiunto.headerGiunto.nome=vett_JointHandles.handles[k].nome;
vett_posOrRelGiunto.posizioneOrientamento.push_back(posOrRelGiunto);
}
else
{
  ROS_WARN("\n Connecting...!\n");
  return;
}
ComauPosOrRel_pub.publish(vett_posOrRelGiunto);
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "NComau_PosOrReL");
  ros::NodeHandle node;
  ros::Subscriber subInfo=node.subscribe("/vrep/info",1,infoCallback);
  ros::Subscriber sub = node.subscribe("Comau_JointHandles", 1000,
  chatterCallback);
  client_groupData=node.serviceClient<vrep_common::simRosGetObjectGroupData>("/
  /vrep/simRosGetObjectGroupData");
  srv_groupData.request.objectType=sim_object_joint_type;
  ComauPosOrRel_pub = node.advertise<tesi::vett_posOrRelGiunto>("/
  Comau_PosOrRel", 1000);
  while (ros::ok()&&simulationRunning)
  {
    ros::spinOnce();
    if(trigger==true)
      LeggiGroupData();
    usleep(5000);
  }
  ROS_INFO("\nSimulation ended! \n");
  return 0;
}
D.2.10  NComau_PosOrAss (NComau_PosOrAss.cpp)

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "ros/ros.h"
#include <iostream>
#include "tesi/ComauMsg.h"
#include "sensor_msgs/JointState.h"
#include <std_msgs/Header.h>
#define new messages
#include "tesi/vett_posOrAssGiunto.h"
#include "tesi/posOrAssGiunto.h"
#include <tesi/JointHandles.h>
#include <tesi/vett_JointHandles.h>
// -----------------
	tesi::vett_posOrAssGiunto vett_posOrAssGiunto;

tesi::posOrAssGiunto posOrAssGiunto;

tesi::vett_JointHandles vett_JointHandles;
// -----------------
#include "vrep_common/VrepInfo.h"
#include "/root/fuerte_workspace/sandbox/vrep/vrep_plugin/include/v_repConst.h"
#include "vrep_common/simRosGetJointState.h"
#include "vrep_common/simRosGetObjectGroupData.h"
tesi::ComauMsg ComauMsg;
ros::Publisher ComauPosOrAss_pub;
bool simulationRunning=true;
float simulationTime=0.0f;

void infoCallback(const vrep_common::VrepInfo::ConstPtr & info)
{
    simulationTime=info->simulationTime.data;
    simulationRunning=(info->simulatorState.data&1)!=0;
}

bool trigger=false;
void chatterCallback(const tesi::vett_JointHandles::ConstPtr & Msg)
{
    vett_JointHandles.handles.clear();
    vett_JointHandles=*Msg;
    trigger=true;
}

ros::ServiceClient client_groupData;
vrep_common::simRosGetObjectGroupData srv_groupData;
void LeggiGroupData()
{
    vett_posOrAssGiunto.posizioneOrientamento.clear();
    srv_groupData.request.dataType=9;
    if(client_groupData.call(srv_groupData))
    {
        for(int k=0;k<7;k++)
        {
            posOrAssGiunto.Posizione_Assoluta[0]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6];
            posOrAssGiunto.Posizione_Assoluta[1]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6+1];
            posOrAssGiunto.Posizione_Assoluta[2]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6+2];
            posOrAssGiunto.Orientamento_Assoluto[0]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6+3];
            posOrAssGiunto.Orientamento_Assoluto[1]=srv_groupData.response.floatData[(vett_JointHandles.handles[k].indice)*6+4];
        }
    }
}
```
D.2.11 Comau threaded child script

```
-- This is a threaded script, and is just an example!
simSetThreadSwitchTiming(2)
simDelegateChildScriptExecution()

SIX = simGetObjectHandle('SIX')
jointHandles = [-1,-1,-1,-1,-1,-1]
for i=1,6,1 do
  jointHandles[i] = simGetObjectHandle('SIX_joint_'..i)
end
-- Set-up some of the RML vectors:
vel=180
accel=40
jerk=80
currentVel = simGetJointVelocity(jointHandles)
maxVel = {vel*math.pi/180, vel*math.pi/180, vel*math.pi/180, vel*math.pi/180, vel*math.pi/180, vel*math.pi/180}
maxAccel = {accel*math.pi/180, accel*math.pi/180, accel*math.pi/180, accel*math.pi/180, accel*math.pi/180, accel*math.pi/180}
maxJerk = {jerk*math.pi/180, jerk*math.pi/180, jerk*math.pi/180, jerk*math.pi/180, jerk*math.pi/180, jerk*math.pi/180}
targetVel = {0,0,0,0,0,0}
```
D.2. ROBOTS

22  -- rotazione primo joint intorno a z
23  targetPos1 = {-170* math.pi /180,0,0,0,0,0}
24  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
25                  maxAccel,maxJerk,targetPos1,targetVel)
26  targetPos1={0,0,0,0,0,0}
27  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
28                  maxAccel,maxJerk,targetPos1,targetVel)
29  targetPos1={170*math.pi/180,0,0,0,0,0}
30  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
31                  maxAccel,maxJerk,targetPos1,targetVel)
32  targetPos1={0,0,0,0,0,0}
33  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
34                  maxAccel,maxJerk,targetPos1,targetVel)
35  --
36  -- rotazione secondo joint intorno a y,
37  targetPos1={0,-85* math.pi /180,0,0,0,0}
38  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
39                  maxAccel,maxJerk,targetPos1,targetVel)
40  targetPos1={0,0,0,0,0,0}
41  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
42                  maxAccel,maxJerk,targetPos1,targetVel)
43  targetPos1={0,150*math.pi/180,0,0,0,0}
44  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
45                  maxAccel,maxJerk,targetPos1,targetVel)
46  targetPos1={0,0,0,0,0,0}
47  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
48                  maxAccel,maxJerk,targetPos1,targetVel)
49  --
50  -- rotazione terzo joint intorno al suo y
51  targetPos1={0,0,170* math.pi /180,0,0,0}
52  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
53                  maxAccel,maxJerk,targetPos1,targetVel)
54  targetPos1={0,0,90* math.pi /180,0,0,0}
55  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
56                  maxAccel,maxJerk,targetPos1,targetVel)
57  --
58  -- rotazione quarto joint intorno al suo z, intorno a x globale
59  targetPos1={0,0,90*math.pi/180,210*math.pi/180,0,0}
60  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
61                  maxAccel,maxJerk,targetPos1,targetVel)
62  targetPos1={0,0,90*math.pi/180,0,0,0}
63  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
64                  maxAccel,maxJerk,targetPos1,targetVel)
65  targetPos1={0,0,90*math.pi/180,-210*math.pi/180,0,0}
66  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
67                  maxAccel,maxJerk,targetPos1,targetVel)
68  targetPos1={0,0,90*math.pi/180,0,0,0}
69  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
70                  maxAccel,maxJerk,targetPos1,targetVel)
71  --
72  -- rotazione quinto joint intorno al suo y
73  targetPos1={0,0,90* math.pi /180,0,-130* math.pi /180,0}
74  simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
75                  maxAccel,maxJerk,targetPos1,targetVel)
76  targetPos1={0,0,90*math.pi/180,0,0,0}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
    maxAccel,maxJerk,targetPos1,targetVel)
targetPos1={0,0,90*math.pi/180,0,130*math.pi/180,0}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
    maxAccel,maxJerk,targetPos1,targetVel)
targetPos1={0,0,90*math.pi/180,0,0,0}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
    maxAccel,maxJerk,targetPos1,targetVel)
--
-- rotazione quinto joint intorno al suo y
targetPos1={0,0,90*math.pi/180,0,0,-270*math.pi/180,0}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
    maxAccel,maxJerk,targetPos1,targetVel)
targetPos1={0,0,90*math.pi/180,0,0,0,0}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
    maxAccel,maxJerk,targetPos1,targetVel)
targetPos1={0,0,90*math.pi/180,0,0,0,0}
simRMLMoveToJointPositions(jointHandles,-1,currentVel,currentAccel,maxVel,
    maxAccel,maxJerk,targetPos1,targetVel)
--
# List of Figures

2.1 The structure of the ROS filesystem of our project. . . . . . . . 8  
2.2 ROS Computation Graph . . . . . . . . . . . . . . . . . . . . . 9  
2.3 V-REP scene screenshot . . . . . . . . . . . . . . . . . . . . . 10  
2.4 three main elements in V-REP . . . . . . . . . . . . . . . . . 11  
2.5 Scene objects . . . . . . . . . . . . . . . . . . . . . . . . . . . 11  
2.6 V-REP calculation modules . . . . . . . . . . . . . . . . . . . . 12  
2.7 V-REP control mechanisms . . . . . . . . . . . . . . . . . . . . 13  

3.1 MERLIN Lab. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17  
3.2 ABB IRB 140 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18  
3.3 COMAU SMART SiX robot . . . . . . . . . . . . . . . . . . . . . . 18  
3.4 KUKA youBot . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19  
3.5 Microsoft Kinect . . . . . . . . . . . . . . . . . . . . . . . . . . . 19  
3.6 Axis 212 PTZ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19  
3.7 Hokuyo URG 04LX UG01 . . . . . . . . . . . . . . . . . . . . . . 20  
3.8 Proximity sensors . . . . . . . . . . . . . . . . . . . . . . . . . . 20  
3.9 ABB IRB140 V-REP model . . . . . . . . . . . . . . . . . . . . . 21  
3.10 KUKA YouBot V-REP model . . . . . . . . . . . . . . . . . . . . 21  
3.11 Laboratorio MERLIN modellato in V-REP . . . . . . . . . . . . . 22  
3.12 Types of proximity sensors . . . . . . . . . . . . . . . . . . . . . 23  
3.13 Proximity sensors distribution. front view . . . . . . . . . . . . 23  
3.14 Proximity sensor dialogue . . . . . . . . . . . . . . . . . . . . . 24  
3.15 Proximity sensors distribution. top view . . . . . . . . . . . . . 24  
3.16 Proximity sensors distribution. lateral view . . . . . . . . . . . 25  
3.17 Example, V-REP Kinect’s filter components . . . . . . . . . . . 26  
3.18 Vision sensor buffers and operations between buffers . . . . . . . 27  
3.19 rangefinder based on vision sensors . . . . . . . . . . . . . . . 27  
3.20 vision sensor based 3D Laser Scanner Fast simulation . . . . . . 28  
3.21 vision sensor based Hokuyo URG 04LX UG01 Fast simulation . . 29  
3.22 proximity sensor based 3D Laser Scanner simulation . . . . . . 30  
3.23 proximity sensor based Hokuyo URG 04LX UG01 simulation . . 31
3.24 Point cloud using Hokuyo URG 04LX UG01 ROS.......................... 32
3.25 Blob detection of a KUKA YouBot........................................... 35

4.1 Graph structure ........................................................................ 39
4.2 robot-subset ............................................................................ 40
4.3 /ComauHandles topic ................................................................ 41
4.4 /IrbHandles topic ..................................................................... 41
4.5 /YouBotHandles topic ............................................................... 41
4.6 [/robot] topic list ....................................................................... 41
4.7 the [/Comau] topics .................................................................. 42
4.8 [/NComau_PosOrAss] and [/NComau_PosOrRel] topics ...................... 43
4.9 sensor-subset ............................................................................ 44
4.10 [/proxSensors] node's topic ......................................................... 45
4.11 Proximity sensor detecting an object ........................................ 45
4.12 /prox_data topic's message ...................................................... 45
4.13 [/telecamera_0/1] topic and message ........................................ 46
4.14 Sensor_msgs::Image message using the image_view package ...... 47
4.15 Sensor_msgs::Image message using the rqt_gui package .......... 47
4.16 Kinect model on V-REP ........................................................... 48
4.17 /vrep/front_scan topic ............................................................. 49
4.18 Final structure of the project ................................................... 50

5.1 Example simulation ................................................................... 52
5.2 List of all topics published by the simulated workspace .............. 52
5.3 /Comau_PosOrAss topic ............................................................... 53
5.4 /Comau_PosOrRel topic ............................................................. 54
5.5 Hokuyo URG 04LX UG01 point cloud of the simulated scene in rviz 54
5.6 Path followed by the operator's model ....................................... 55
5.7 Blob detection using the two cameras ....................................... 56
5.8 /vrep/Blob_Camera_1_X and /vrep/Blob_Camera_1_Y topics ......... 56
5.9 The generated .csv files ............................................................. 58
5.10 Blob detection displayed in Matlab .......................................... 59
5.11 Irb joints absolute position ...................................................... 59
5.12 Irb Joint 1 .............................................................................. 60
5.13 Irb Joint 2 .............................................................................. 60
5.14 Irb Joint 3 .............................................................................. 60
5.15 Irb Joint 4 .............................................................................. 61
5.16 Irb Joint 5 .............................................................................. 61
5.17 Irb Joint 6 .............................................................................. 61
5.18 Comau Joint 1 ......................................................................... 62
5.19 Comau Joint 2 ......................................................................... 62
5.20 Comau Joint 3 .................................................. 62
5.21 Comau Joint 4 .................................................. 63
5.22 Comau Joint 5 .................................................. 63
5.23 Comau Joint 6 .................................................. 63
5.24 Comau Joint 7 .................................................. 64
5.25 Controlling a robot from the V-REP simulator ................... 64
5.26 ROS Networksetup ............................................. 65
5.27 From the workspace to the simulator ................................ 65

A.1 ROS workspace structure ....................................... 69
A.2 Struttura del workspace ROS con plugin vrep .................. 71
A.3 The files that we need to modify before building the plugins .... 71

B.1 Example 1: V-REP publisher and subscriber ..................... 75
B.2 Example 2: Control a V-REP object via a ROS node ............. 77
B.3 Example 3: Message exchange in both ways ...................... 80
B.4 Example 4: create ROS and V-REP nodes directly from ROS .... 82
Bibliography


http://www.hokuyo-aut.jp/02sensor/07scanner/download/products/urg-04lx-ug01/

http://wiki.ros.org/msg

http://wiki.ros.org/srv

http://wiki.ros.org/rosbuilt/Manifest

[18] C. Robotics, “Ros api list (alphabetical order).”