# ROS analysis

Giulia Brugnatti
giulia.brugnatti@studio.unibo.it

January 2021

**Key Words**
ROS
Navigation Stack
Turtlebot3
Autonomous Navigation
Multiple robots navigation
Gazebo
RViz
RQt

# Contents

# List of Figures

# Chapter 1

# Project introduction

## 1.1 Overview

The project aims to study ROS middleware providing accurate documentation of its architecture, primary functionalities, and building blocks besides illustrating some meaningful study cases of the framework.

## 1.2 Structure

The report is made up of five main parts: which span from the general description of the main features of ROS, the introduction of ROS Navigation Stack, to some application examples.

The first chapter aims to provide a brief introduction to the work, aside from a discussion about its primary goals and requirements.

The second and third parts will focus on ROS middleware; illustrating, bit by bit, its core features, besides a short presentation of some relevant tools used in the ROS environment for simulation.

Fourthly, ROS tools for simulation will be presented, along with a précis of their main characteristics and usage.

The fifth chapter illustrates a dense description of the Navigation Stack's most significant concepts.

Next, it is presented a concise overview of the robot platform used in the final scenario, along with some reasons that justify the choice.

The seventh and eighth chapters are devoted to the use cases; the first one is centered on ROS basic building blocks, whereas the latter focuses on ROS robotics application.

Finally, the last section is the bottom line of the project, presenting some final consideration about ROS environment, in the light of the analysis carried out in the former chapters, as well as a discussion about future works.

## 1.3 Goal

The project aims to create detailed documentation, besides providing a step-by-step guide to experiment on ROS primary elements on one hand and Navigation Stack on the other. Indeed, the first use case chapter focuses on understanding how ROS core components can communicate within the platform, whereas the second centers on Navigation Stack. The purpose of that latter section is to introduce one of the most remarkable robot duties: the navigation task. Therefore, an examination of the primary methods to accomplish robot exploration tasks will be illustrated.

It is crucial to underline, that the project aims not to create a complex robot application, although its target is to provide and analyze some not trivial examples of ROS usage to understand the platform working process; robots introduction along these lines are not that tools used to explain the aforementioned ROS operation.

## 1.4 Requirements

The project study case is based on ROS standard packages, whereas this report does not provide a complete list of the packages that must be installed; this is the reason why it is recommended to refer to the project `README.md` file for that kind of information. Furthermore, the git-lab package contains the folders needed for the execution of the last use-case part; therefore, downloading the project at https://gitlab.com/pika-lab/courses/ds/projects/ds-project-brugnatti-ay2021 is kindly suggested.

Additional environment requirements are discussed in Chapter 7.

# Chapter 2

# ROS Introduction

## 2.1 Introduction

The Robot Operating System, better known as ROS, is a collection of libraries, tools, and conventions whose main aim is to simplify the creation and management of robust robot software for a wide variety of robot platforms, sensors, and effectors.

## 2.2 History

ROS development dated back to 2007, when in the Stanford Artificial Intelligence Laboratory, Morgan Quigley[6], Eric Berger and Andre Y. Ng[86] creates a library to support the STAIR[1] project and the PR[2].

That library was named Switchyard, and its primary purpose was to "supports parallel processing through message passing along a user-defined, task-specific graph of connections between software modules"[41]. Switchyard was such a puissant idea that in 2009 the Stanford team, with the collaboration of the Willow Garage[3], presented *ROS: An Open-Source Robot Operating System* at the *IEEE International Conference on Robotics and Automation* (ICRA) in Japan.

Next, in 2010 the first stable implementation of the framework was released, and since then ROS ecosystem has not stopped to evolve becoming more elaborated and wider day by day.

[1] STAIR: STanford Artificial Intelligence Robot
[2] PR: Personal Robots
[3] Further information about Willow Garage research lab are in the Appendix section10.2

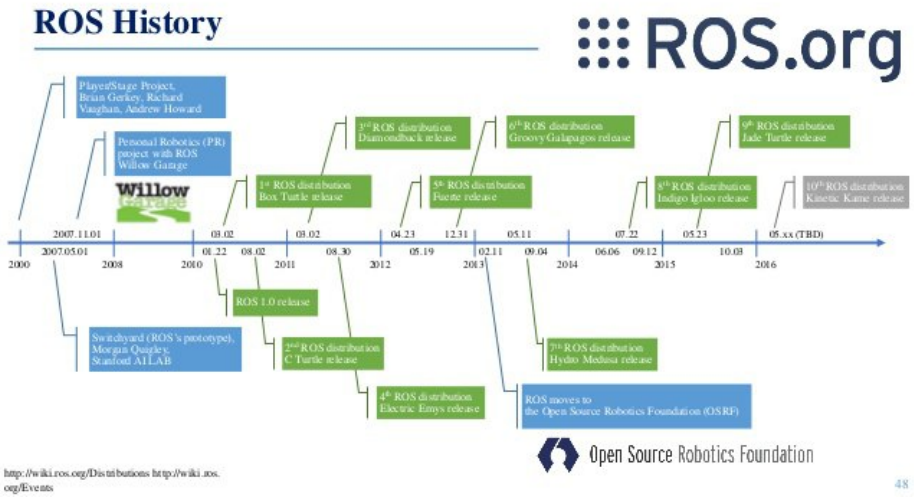


Figure 2.1: ROS history

## 2.3 Philosophy

ROS initial goals can be sum up in the following points:

- **Peer-to-peer**: ROS systems are made up of several processes, possibly on different hosts, linked at runtime in a peer-to-peer topology. This latter arrangement needs some kind of lookup mechanism to allow processes to find each other at runtime[4].
- **Tools-based**: ROS has a microkernel design, where numbers of small tools are used to build and execute the different ROS elements[5]
- **Multi-lingual**: ROS is designed to be language-neutral; supporting several languages like C++, Python, Octave, Java, Lua, and LISP, with other language ports in various states of completion.
- **Thin**: ROS build system performs modular builds inside the source code tree; placing virtually all complexity in libraries, and only creating small executables which expose library functionality to ROS, allows for easier code extraction, reuse, and testing beyond its original intent.
- **Free and Open-Source**: ROS source code is publicly available[42].

[4]ROS provide a specific tool for this purpose: the master, which will be analyzed in the next chapter.

[5]ROS developers believed that the loss in efficiency caused by the choice of not constructing a monolithic development and runtime environment is more than offset by the gains instability and complexity management.

### 2.3.1 What ROS is not

ROS is an open-source framework that provides useful tools and libraries to help software developers in the creation of robotics application, and it should not be confused with:

- a **programming language**; indeed ROS is not a programming language ,but it supports different languages in C++, Python, and Lisp.
- an **operating system**, as exposed in the former section, even though it shares some features with operating systems ROS is only a meta-operating system. In the next table2.2 the major differences between ROS and a common operating system are shown.
- an **integrated development environment**; although ROS provides tools for debugging, implementation and simulation is not an IDE.
- a **library**; ROS does include client libraries, but also a central server, a build system, a set of command-line tools, and graphical tools[34].

What Makes the Difference...?

| Conventional OS | ROS |
|---|---|
| Explicitly a general purpose OS | Exclusive for Robotic Platform |
| Ortho-Operational | Meta-Operational |
| Native Language Programming | Language-independent architecture |
| Sequential Architecture | Asynchronous Distributed architecture |
| Programming IDE | Software Frameworks |
| Propriety/Open-Source | Open-source under BSD license |
| Heavily coded frameworks | ROS frameworks are very light |
| Programs | Nodes |
| Communication | Messages |
| Kernel is Included | Kernelless |

Figure 2.2: Table exposing key differences between ROS and an operating system

### 2.3.2 Why ROS

The most relevant features that make ROS the de facto standard in robotics software[6] outlined as follows:

- **Distributed computation**: many modern robot systems rely on software that spans several processes and runs across several different comput-

ers. ROS provides two considerably basic mechanisms for communication between processes that may or may not live on the same computer.

- **Software reuse**: the rapid progress of robotics research has resulted in a growing collection of efficient algorithms for common tasks (i.e navigation, mapping, etc.), however, their existence is only useful if there is a way to apply them in a new context without needing to re-implement them. In this respect, ROS can help to prevent to *reinventing the wheel* offering a set of software for operating robots that can be used as *off the shelf* programs or be extended.
- **High-end capabilities** ROS has ready for use highly configurable capabilities, for instance, *SLAM*[6] and *AMCL*[7] packages that can directly be utilize in robots software.
- **Tons of tools** ROS is comes with several tools, as `rqt_gui`, *RViz* and *Gazebo*, for debugging, visualizing, and performing simulation.
- **Support high-end sensors and actuators** ROS has a multitude of device drivers and interface packages of various sensors and actuators in robotics (i.e *Velodyne-LIDAR*, *Laser scanners*, etc.).
- **Inter-platform operability** The ROS message-passing middleware allows communicating among different nodes that can be programmed in any language that has ROS client libraries.
- **Modularity** One of the issues that can occur in most standalone robotic applications is, if any of the threads of the main code crash, the entire robot application can stop. ROS overcome this limitation on one hand by making the programmer writing different nodes for each process in this way if one node crashes, the system can still work, and on the other by providing robust methods to resume operation even if any sensors or motors are dead.
- **Concurrent resource handling** ROS simplifies the operation of handling a hardware resource by more than two processes by making it possible to access the devices using ROS topics from the ROS drivers. In this way multiple ROS nodes can subscribe to the same message from a specific driver and each node can perform different functionalities. This feature can reduce the complexity in computation as well as increase the debug-ability of the whole system.
- **Active community** One of the most relevant factors, that must be considered while choosing to use a library or software framework from an open-source community, is if it is supported by a developer community. In ROS, the support community is highly active, besides there is a web portal to handle the support queries from the users[8][34][26].

[6]SLAM: Simultaneous Localization and Mapping
[7]AMCL: Adaptive Monte Carlo Localization
[8]ROS Answers site: (*http://answers.ros.org*)

## 2.4 ROS 1 vs ROS 2

### 2.4.1 Overture

In 2017, besides the former ROS version (ROS 1), another ROS project was released. That new version was born from the necessity to overcome some of the limits the former platform presented. Notably, among the requirements for the newborn framework there were: robot security, real-time control along with an increase in the distributed processing.

### 2.4.2 Characteristics

The main differences between the ROS version will be examined in the following paragraph[9] :

- **Language**: ROS 1 core target C++03 and python 2, however it does not use C++11 in its API, on the contrary, ROS 2 use C++11 extensively along with some part of C++14 and it also supported the latest python version.
- **Platform** ROS 1 is only being tested on Ubuntu and supported by the community on other Linux flavors as well as OS X; however, ROS 2 is being tested and supported on Ubuntu Xenial, OS X El Capitan as well as Windows 10.
- **Reusing existing middleware** ROS 1 uses:
  - a custom serialization format,
  - a custom transport protocol,
  - a custom central discovery mechanism

  However, ROS 2 uses an abstract middleware interface to provide serialization, transport, and discovery; currently, all implementations of that interface are built upon the DDS standard. This feature enables ROS 2 to supply much different Quality of Service policies that ameliorate communication over diverse networks.
- **Build system** ROS 2 gives users the possibility to support several build systems besides CMake.
- **Process** ROS 2 allow users to create multiple nodes in the same process[13][9].

[9] If one is interested in all the differences between the two projects, one can retrieve all the information about the changes at the following link: https://design.ros2.org/articles/changes.html

### 2.4.3 Why a new project?

Despite the fact, the group considered enhancing the current version of ROS after a risk-benefit analysis they concluded that *given the intrusive nature of the changes that would be required to achieve the benefits that we are seeking, there is too much risk associated with changing the current ROS system that is relied upon by so many people*[13].

They aimed to let ROS 1 continue to work uninfluenced by the development of ROS 2; for this reason, they have chosen to develop it as a *parallel set of packages that can be installed alongside keep working and interoperate with ROS 1*[13].

### 2.4.4 Choice

Since this project aims to provide documentation for ROS beginners along with some general examples of ROS case uses, there is no need to value real-time control as well as robot security; therefore, the choice to use ROS 1 as the target version. Furthermore, the former version is especially recommended for students as well as ROS general developers[9].

# Chapter 3

# ROS Architecture

## 3.1 Overview

The structure of ROS is made up of three main level concepts:

- the **File system level**
- the **Computational Graph level**
- the **Community level**

Those levels will be discussed with further details in the following sections, along with the two types of naming structures, **Package Resource Names** and **Graph Resource Names**, that ROS defined.

## 3.2 File system

Like operating system, ROS file system organizes its files and folder into a hierarchical structure on disc as shown by the following graph:

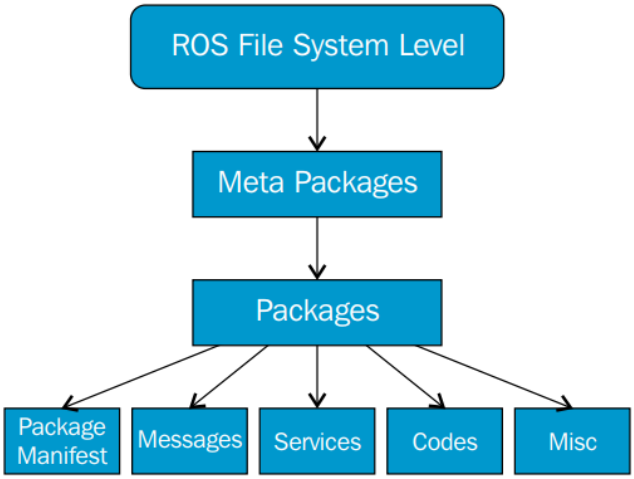


Figure 3.1: ROS file system's graph

In the following paragraphs, will be given a brief delineation of each element of the former diagram[26].

### 3.2.1 Meta packages

Meta packages are specialized structures whose aim is to depict a group of packages that have a special purpose.

Unlikely, other kind of packages they contain only one file: the `package.xml` file. In the `package.xml` file, the meta package contains an export tag, as follows:

```
<export>
    <metapackage/>
</export>
```

Also, in meta packages, there are no `<buildtool_depend>` dependencies for Catkin[1], there are only `<run_depend>` dependencies, which are the packages grouped in the meta package.[26]

**Meta packages manifest** : it is an `.xml` file which contains the main information about the package (author, license, dependencies, etc.) besides including packages inside as runtime dependencies[26].

### 3.2.2 Packages

ROS packages are the most basic unit of the ROS software; they contain ROS runtime processes (nodes), libraries, configuration files, and so forth, which are

[1] A description of Catkin build system can be found in chapter 10.2 of the Appendix section

organized together as a single unit. Packages are the atomic build item and release item in the ROS software.[34]

| Directory | Explanation |
|---|---|
| include/ | C++ include headers |
| src/ | Source files |
| msg/ | Folder containing Message (msg) types |
| srv/ | Folder containing Service (srv) types |
| launch/ | Folder containing launch files |
| package.xml | The package manifest |
| CMakeLists.txt | CMake build file |

Figure 3.2: ROS package's common files and directories

**Package manifest** : it is similar to the meta-package manifest; however, unlike it, the package manifest can not include packages inside it as runtime dependencies[26].

### 3.2.3 Messages

ROS Processes communicate with each other by publishing messages to topics. A message is a basic data structure, comprising typed fields which support standard primitive types as well as arrays of primitive types and constants. The following table (3.3) illustrates some of the built-in field types that can be used in a message: Furthermore messages can include arbitrarily nested structures and arrays[58] as well as being created ex-novo by users. The extension of the message file is `.msg`, besides they can be defined as follow:

```
int32 number
string name
float32 speed
```

The message definition can consist of two types: fields and constants. The field is split into field types and field names; field types are the data type of the transmitting message, and field name is the name of it, whereas constants define a constant value in the message file.

| Primitive type | Serialization | C++ | Python |
|---|---|---|---|
| bool(1) | unsigned 8-bit int | uint8_t(2) | bool |
| int8 | signed 8-bit int | int8_t | int |
| uint8 | unsigned 8-bit int | uint8_t | int (3) |
| int16 | signed 16-bit int | int16_t | int |
| uint16 | unsigned 16-bit int | uint16_t | int |
| int32 | signed 32-bit int | int32_t | int |
| uint32 | unsigned 32-bit int | uint32_t | int |
| int64 | signed 64-bit int | int64_t | long |
| uint64 | unsigned 64-bit int | uint64_t | long |
| float32 | 32-bit IEEE float | float | float |
| float64 | 64-bit IEEE float | double | float |
| string | ascii string(4) | std::string | string |
| time | secs/nsecs unsigned 32-bit ints | ros::Time | rospy.Time |
| duration | secs/nsecs signed 32-bit ints | ros::Duration | rospy.Duration |

Figure 3.3: ROS builtin types

### 3.2.4 Services

ROS services are a kind of request/response interaction between processes. The reply and request data types can be defined inside the srv folder inside the package (`my_package/srv/MyServiceType.srv`)[26].

## 3.3 Computation graph level

ROS uses a network of processes[2] to implement its computation; the peer-to-peer computation network of ROS processes is called *Computation Graph*. The basic building blocks of the Computation Graph are: *nodes*, *Master*, *Parameter Server*, *messages*, *topics*, *services*, and *bags*, all of which provide data to the Graph in different ways.

### 3.3.1 Nodes

Nodes are processes that perform computation; each of them is written using ROS client libraries which are also used to implement different types of communication techniques in nodes. Notably, one node can communicate with other nodes using *topics*, *services*, and *parameters*. To identify a node in the system a name should be assigned to it. The `rosnode` command can be used to get introspect ROS nodes. The list of `rosnode` usage can be summarized as follows:

- `$ rosnode info [node_name]`: it prints the information about the node
- `$ rosnode kill [node_name]`: it teardowns a running node
- `$ rosnode list`: it list the running nodes

[2]Processes are called *nodes* in ROS environment.

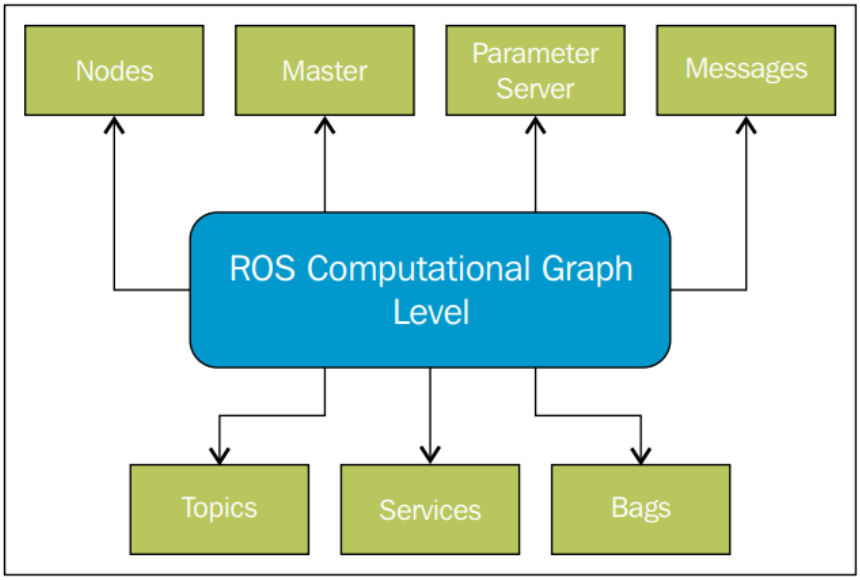


Figure 3.4: ROS graph level

- `$ rosnode machine [machine_name]`: it lists the nodes running on a particular machine or a list of machines
- `$ rosnode ping`: it checks the connectivity of a node
- `$ rosnode cleanup`: it purges the registration of unreachable nodes[26]

### 3.3.2 Master

The ROS Master provides name registration and lookup system nodes; therefore, nodes would not be able to find each other, exchange messages, or invoke services without it. ROS requires only one master per system.

ROS Master has many features in common with a DNS server; indeed when a node starts in the ROS system, it looks for the master and registers its name in it. In that way the Master has the details of all nodes currently running on the ROS system, and when any of these change, it generates a call-back and update with the latest details.

ROS client libraries, as `roscpp` and `rospy` for instance, are used to write nodes. Those libraries interact with the Master using `XMLRPC` (XML Remote Procedure Call) based APIs, which act as the backend of the ROS system APIs. ROS master's IP and port are stored in an environment variable called

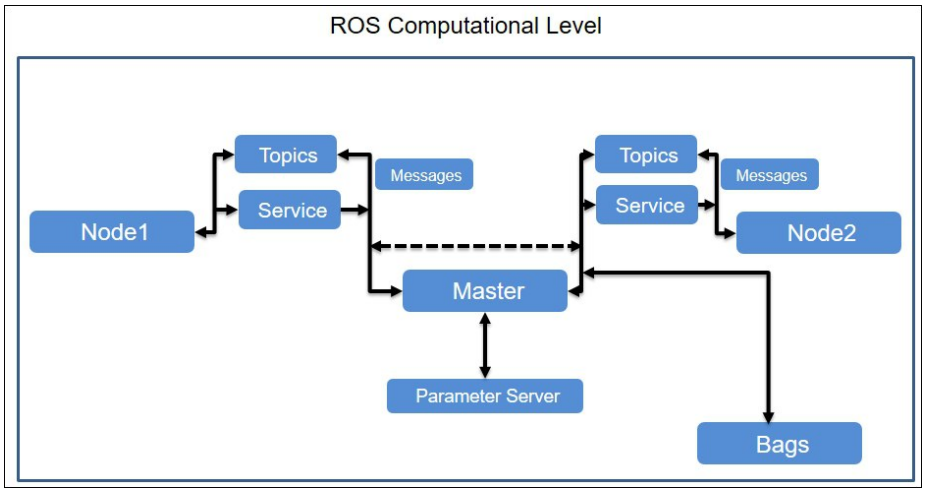


Figure 3.5: ROS computational communication's graph

`ROS_MASTER_URI`, that latter is used by nodes find ROS Master and start the communication.

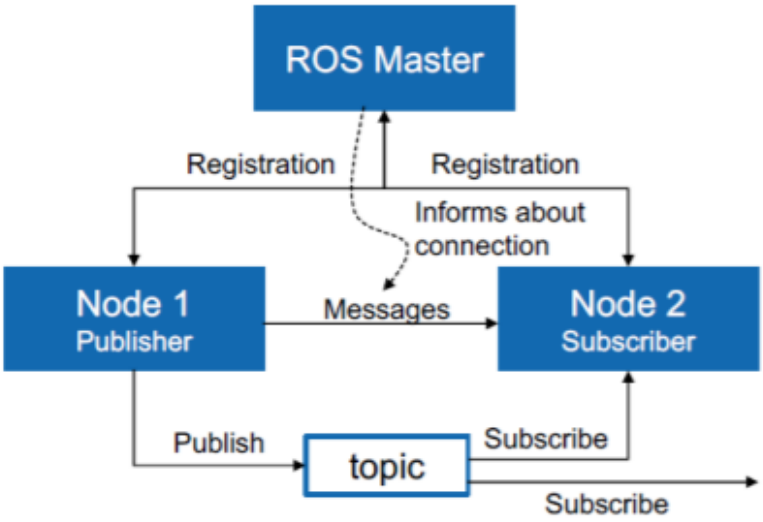


Figure 3.6: ROS: master-client registration's graph

**Registration process**

A node starts publishing a topic, providing details, like name and data type, about it to ROS Master as soon as the Master receives those data will check if nodes subscribed to that topic are present in the system. If any nodes are subscribed to the same topic, the publisher details will be shared by ROS Master with them, after that procedure has been completed, the Subscriber and Publishers nodes will interconnect using the *TCPROS protocol*(mainly based on TCP/IP sockets). In these terms, ROS Master's primary purpose is to connect the nodes, when it performs that operation it has no longer a role in controlling them[26].

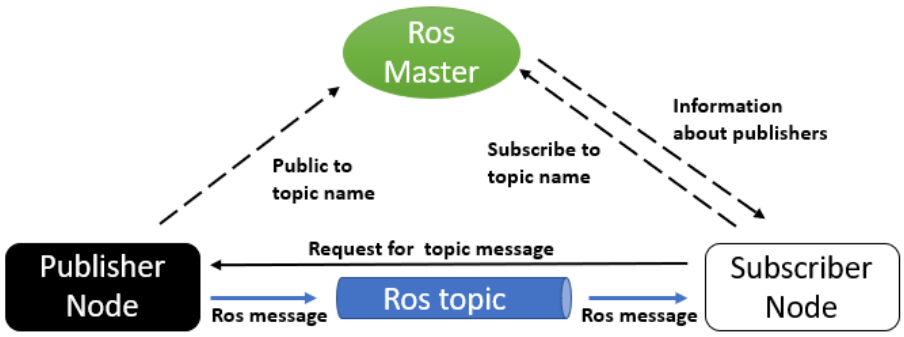


Figure 3.7: ROS publish-subscribe's graph

#### 3.3.2.1 Parameter Server

The parameter server is the part of the Master who provides a mechanism to save by key the data in a central location accessible by all nodes. In this way, nodes can modify the information present in the server[26].

### 3.3.3 Messages

Nodes communicate (peer-to-peer) with each other passing messages to topics.

ROS naming convention is used to name message types: package name + message name; thus ROS can automatically generate source code to use messages in a ROS node, both in C++ and python.

Nodes can also communicate using service calls; indeed, services too are messages, especially the service message definitions are defined inside the `.srv` file.

Furthermore, apart from the message data type, an MD5 checksum comparison is used by ROS to confirm whether the publisher and subscriber exchange

the same message data types.

ROS inbuilt tool to get information about ROS messages is called `rosmsg`; it can be used with a set of parameters to perform different operation as follows:

- `$ rosmsg show [message]`: it shows the message description
- `$ rosmsg list`: it lists all messages
- `$ rosmsg md5 [message]`: it displays `md5sum`[3] of a message
- `$ rosmsg package [package_name]`: it lists messages in a package
- `$ rosmsg packages [package_1] [package_2]`: it lists packages that contain messages[26]

### 3.3.4 Topics

ROS messages are routed via a transport system with publish-subscribe semantics using strongly typed message buses called topics. Nodes send messages through a topic, by publishing it; topics have a name that is used to identify the content of the message, therefore a node that is interested in a certain kind of data will subscribe to the appropriate topic. Nevertheless, since the publish-subscribe model implements many-to-many communication, it is not suitable for request/reply interactions; to implement that kind of communication, one has to use ROS services. To decouple the production of data from its consumption publishers and subscribers are not aware of each others' existence. That is the reason why in a ROS system one could have multiple concurrent publishers and subscribers for a single topic, a single node may publish and/or subscribe to several topics, and also a node could even subscribe to a topic that might not have any publisher.

The ROS nodes communicate with topics using TCP/IP-based transport known as `TCPROS`[4]. This method is the default transport method used in ROS. Nevertheless, ROS also provides another type of communication that is called `UDPROS`, that latter has low-latency, loose transport, and is only suited for teleoperation[26].

Likewise for messages, also for topics exists a tool to retrieve information about them: that last is called `rostopic` and its main usage can be summarized as follows:

- `$ rostopic bw /topic`: it displays the bandwidth used by the given topic
- `$ rostopic echo /topic`: it prints the content of the given topic
- `$ rostopic find /message_type`: it finds topics using the given message type

[3] MD5 is a computer program that calculates and verifies 128-bit MD5 hashes, as described in RFC 1321. The MD5 hash functions as a compact digital fingerprint of a file[87]

[4] TCPROS: is a transport layer for ROS Messages and Services. It uses standard TCP/IP sockets for transporting message data. Inbound connections are received via a TCP Server Socket with a header containing message data type and routing information.[64]

- `$ rostopic hz /topic`: it shows the publishing rate of the given topic
- `$ rostopic info /topic`: it prints information about an active topic
- `$ rostopic list`: This command will list all active topics in the ROS system
- `$ rostopic pub /topic message_type args`: it publishes a value to a topic with a message type
- `$ rostopic type /topic` : it displays the message type of the given topic[26].

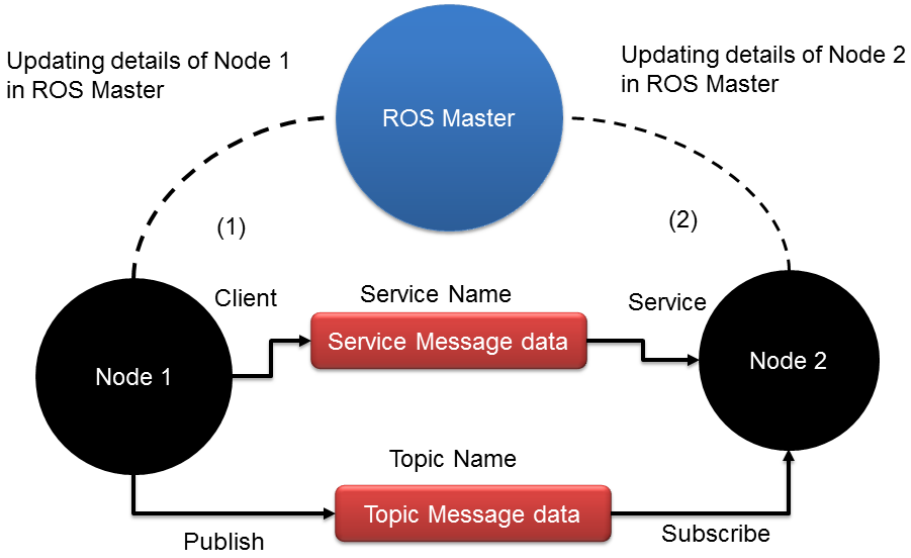


Figure 3.8: ROS client-server interaction's graph focused on topics and services

### 3.3.5 Services

The ROS services are a type of request-response communication between ROS nodes. One node will send a request and wait until it gets a response from the other. analogously to the message definition, a service description language is used to define the ROS service types (`.srv`).

```
#Request message type
string str
---
#Response message type
string str
```

Listing 3.1: Example of service description format

As can be seen, one can define a service definition that contains two parts: one is for requests, and the other is for responses. Indeed in the example provided the first section of the service is the message type of request, and it is separated by "—" from the following section that is the message type of response. In ROS services, one node acts as a ROS server in which the service client can request the service from the server. If the server completes the service routine, it will send the results to the service client. As well as ROS messages, also ROS services use MD5 checksum to control service message correctness.

There are two tools to get information about the ROS service: the first tool is `rossrv`, which is similar to *rosmsg* and is used to get information about service types, whereas the second is *rosservice*, is used to list and query about the running ROS services.

`rosservice` usages can be sum up as follows:

- `$ rosservice call /service args`: it calls the service using the given arguments
- `$ rosservice find service_type`: it finds services in the given service type
- `$ rosservice info /services`: it prints information about the given service
- `$ rosservice list`: it lists the active services running on the system
- `$ rosservice type /service`: it prints the service type of a given service
- `$ rosservice uri /service`: it prints the service `ROSRPC URI`[26]

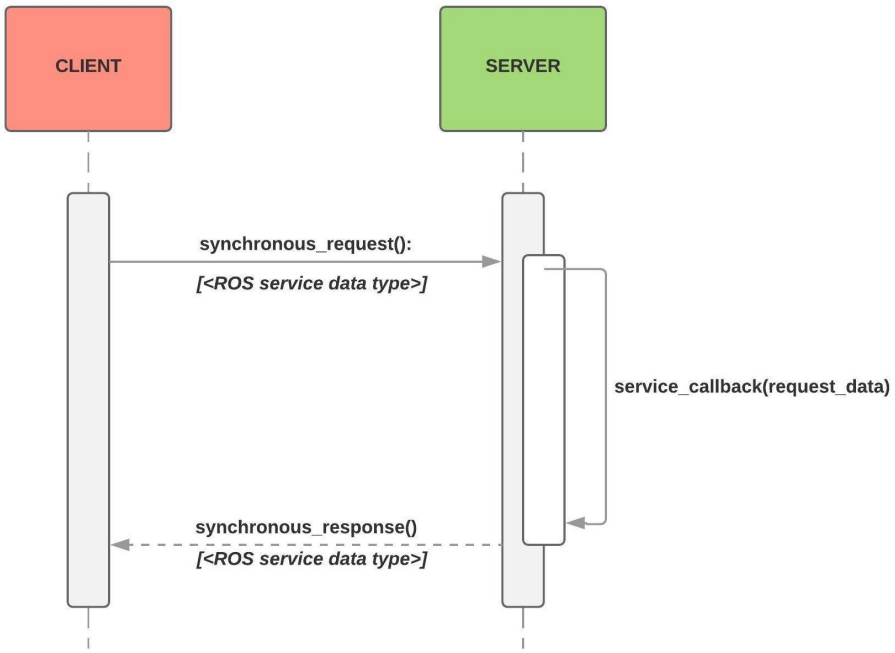


Figure 3.9: ROS client-server synchronous interaction's diagram

### 3.3.6 Bags

ROS uses a standard format called Bags to save and playback ROS message data; notably, a bag file stores ROS message data from topics and services. Bag files are defined by files having `.bag` extension. Data logging is rosbag main purpose; indeed, the robot information can be logged and visualized online whereas processed offline.

The rosbag command is used to work with rosbag files. Here are the commands to record and playback a bag file:

- `$ rosbag record [topic_1] [topic_2] -o [bag_name]`: This command will record the given topics into a bag file that is given in the command. We can also record all topics using the -a argument.
- `$ rosbag play [bag_name]`: This will playback the existing bag file.There is a GUI tool to handle record and playback of bag files called `rqt_bag`[26].

## 3.4 Community level

The community-level purpose is to enable communities to exchange software and knowledge. ROS has three different types of resources that habilitate the

aforementioned process: wiki, distributions, and repositories.

### 3.4.1 Wiki

The ROS community Wiki is the primary forum for documenting data about ROS; it provides a ways for anyone to contribute to ROS ecosystem by uploading ones own documentation, suggesting corrections or updates, as well as writing tutorials, and so forth.

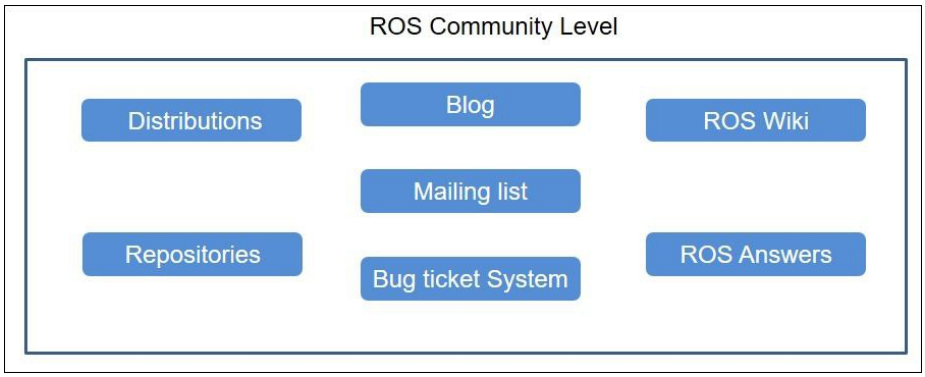


Figure 3.10: ROS Community level's basic components

### 3.4.2 Distributions

ROS distributions are a collection of versioned meta-packages that can be installed; they maintain consistent versions across a set of software[34]. Similar to Ubuntu and Android, ROS has been naming their versions to start in alphabetical order; for instance, the Kinetic Kame version is the 11th version thus starting with the alphabet K. Moreover, each version has an illustration in the form of a poster with a turtle icon, which is also used in the official simulation tutorial called *turtlesim*[5]. The turtle symbol for ROS was stemmed from the educational programming language Logo10.2. Indeed in 1969, a turtle robot was developed by Papert10.2 team using Logo, the aforementioned device was able to move on the floor drawing pictures according to the instructions given by the computer. Based on this robot, ROS *turtlesim* was developed and later also Turtlebot robots6[40].

[5]Turtlesim simulator is analyzed in the use case section 8.2

| Distro | Release date | Poster | *Tuturtle*, turtle in tutorial | EOL date |
|---|---|---|---|---|
| ROS Noetic Ninjemys (**Recommended**) | May 23rd, 2020 |  |  | May, 2025 (Focal EOL) |
| ROS Melodic Morenia | May 23rd, 2018 |  |  | May, 2023 (Bionic EOL) |
| ROS Lunar Loggerhead | May 23rd, 2017 |  |  | May, 2019 |
| ROS Kinetic Kame | May 23rd, 2016 |  |  | April, 2021 (Xenial EOL) |
| ROS Jade Turtle | May 23rd, 2015 |  |  | May, 2017 |
| ROS Indigo Igloo | July 22nd, 2014 |  |  | April, 2019 (Trusty EOL) |
| ROS Hydro Medusa | September 4th, 2013 |  |  | May, 2015 |
| ROS Groovy Galapagos | December 31, 2012 |  |  | July, 2014 |
| ROS Fuerte Turtle | April 23, 2012 |  |  | -- |
| ROS Electric Emys | August 30, 2011 |  |  | -- |
| ROS Diamondback | March 2, 2011 |  |  | -- |
| ROS C Turtle | August 2, 2010 |  |  | -- |
| ROS Box Turtle | March 2, 2010 |  |  | -- |

Figure 3.11: ROS distributions

### 3.4.3 Repositories

ROS depends on a federated network of code repositories, where several institutes can elaborate and release their robot software components[26].

## 3.5 Names

### 3.5.1 Graph Resource Names

Graph Resource Names provides a hierarchical naming structure that is used for all resources in a ROS Computation Graph. It is a significant technique for providing encapsulation: each resource can create resources within their namespace, and they can access resources within or above their namespace.

To facilitate programming names are resolved relatively; thus, resources are unaware of the namespace in which they are located; this is the reason why nodes that work together can be written as if they are all in the top-level namespace[26].

### 3.5.2 Package Resource Names

To streamline the procedure of referring to files and data types on disk Package Resource, besides File System-Level concepts are used. Package Resource Names are basic structures made up of the name of the Package that the resource is aside from the name of the resource. They can be used to refers to message types, service types, as well as, nodes. For instance, the name `std_msgs/String` refers to the "String" message type in the `std_msgs` Package[26].

## 3.6 Building ROS blocks from scratch

This latter section of the chapter will provide basic examples about the creation of some ROS building blocks from scratch; especially it will be shown how to create, build and organize a package in a custom workspace along with the creation of user-defined messages and nodes.

### 3.6.1 Workspace creation

ROS guidelines suggest a specific layout for code organization; for this reason, the first element to create before writing code for a ROS system is the `workspace`. A ROS workspace is a special folder inside which users will actively develop; to initialize a folder as a ROS workspace it is necessary to create a `src`[6] directory inside it and executing the `catkin_make`[7] command as follow:

[6] catkin_make command will not be able to initialize the workspace without an `src` folder inside it.

[7] Catkin build system is described in the section 10.2

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

mkdir -p ~/example_ws/src
cd ~/example_ws/
catkin_make

source devel/setup.bash
```

Listing 3.2: Commands to create a ROS workspace

If the `catkin_make` command worked two new folders should appear in the new workspace: the `devel` and `build`[69] folders.

The source space (`src`) is the folder is where Catkin will be expected to look for packages when building. This folder is easily identified as it is where the toplevel.cmake is linked from the catkin project. Each catkin project desired to be compiled from source should be checked out into subdirectories inside this directory. Packages are found recursively so they do not have to be direct subfolders.

The build space (`build`) is the folder in which `cmake` is invoked and generates artifacts such as the `CMakeCache`.

The development space (`devel`) is where Catkin generates the binaries and runtime libraries that are executable before installation. After the build step, inside this folder is expected everything needed to run nodes in packages that have been built. The development space can not be a folder that contains ROS packages in subfolders[49].

It is valuable to notice that in the proposed list of commands are executed also two `source` commands, which are used by ROS to properly locate the files, if ROS directories are not properly sourced it will launch an error on the terminal.

### 3.6.2 Package creation

To create a new package in the workspace one should move inside the `src` folder and execute the following commands:

```
catkin_create_pkg example_pkg std_msgs rospy roscpp
```

Listing 3.3: Commands to create a ROS package

For this purpose `catkin_create_pkg` command is used to properly initialize the folder as a package creating the: `CMakeLists.txt` and `package.xml` files with the first level dependencies: `std_msgs`, `rospy` and `roscpp`. Since ROS is not notified when a workspace is modified, for example by adding new packages, it is necessary to rebuild it and re-source the setup file to inform it that a new package is available in the `example_ws`[68][67].

### 3.6.3 Message creation

At this point, to create a new message it is just necessary to create a new file with `.msg` extension; however, ROS best-practice rules require locating a

new message inside a proper directory called `msg`. To do that, one could move inside the `example_pkg` folder and only execute the following commands:

```
mkdir msg

echo "int64 num" > msg/Num.msg
```

Listing 3.4: Commands to create a ROS message

The last command, create a message with a line only, however one could decide to create a more complex file by adding multiple elements. Since the package dependencies are not update automatically, one should add the following line to the `package.xml` file:

```
  <build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
```

Listing 3.5: Commands to create a ROS message

Furthermore, also the `CMakeLists.txt` file should be updated by adding the `message_generation` dependency among the catkin required components in order to be able to generate messages. Finally, one should also modify the `generate_messages` block of code by removing the # symbols on one hand and replacing the `Message*.msg` files with the name of the message created in the previous paragraph:

```
add_message_files(
  FILES
  Num.msg
)
```

Listing 3.6: generated_message block

Now, to check if ROS can see the message one could run:

```
rosmsg show example_pkg/Num
```

The output should be[8]:

```
int64 num
```

The same procedure can be used to create a srv[70].

### 3.6.4 Node creation

This latter section illustrates how to create ROS nodes (a publisher and a subscriber) using scripts. ROS tutorials provide a basic implementation of publisher and subscriber nodes in `Python` and `C++`, in this section will be used the `Python` one. Before continuing it is important to notice that because of ROS package convention the two nodes' scripts must be located in a `scripts` folder.

[8] If ROS cannot find the new message it could be useful to execute catkin_make again.

**Publisher** : To download the publisher file the following command should be launched in the `scripts` directory:

```
wget https://raw.github.com/ros/ros_tutorials/kinetic-devel/
    rospy_tutorials/001_talker_listener/talker.py
```

Next, it is necessary to make it executable:

```
chmod +x talker.py
```

The downloaded script should appear as follow:

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Listing 3.7: Python publisher script

The declaration at the top of the script is to make sure that is executed as a python script. In the following line, two imports are found, the first one is required to write ROS nodes, while the second is needed to reuse the `std_msgs/String` message type[9]. Next, a python function is defined to implement the publisher; especially it declares a node that is publishing to the chatter topic using the message type String[10]. The `queue_size` argument limits the amount of queued messages if any subscriber is not receiving them fast enough. The subsequent line provides the node initialization: specifying its name to the system [11], and making it unique initializing the anonymous parameter as *true*[12]. Before defining the node behavior, a `rate` object is created to execute the node loop at a given rate. The loop is the core of the script, as it defines the task of the publisher node: in this case, to call `pub.publish(hello_str)` that publishes a string to the chatter topic until a shutdown condition is provided. The loop calls `rate.sleep()`, which sleeps just long enough to maintain the desired rate through the loop.

[9] It is not mandatory to use ROS builtin message types, indeed one could also choose to use user-defined messages importing them from a custom package.

[10] String here is the class `std_msgs.msg.String`

[11] The name must be a base name, i.e. it cannot contain any slashes "/".

[12] `anonymous =True` ensures that your node has a unique name by adding random numbers to the end of NAME.

This loop also calls `rospy.loginfo(str)`, which performs triple-duty: the messages get printed to screen, it gets written to the Node's log file, and it gets written to rosout. Finally, the last lines of the script consist in a standard Python `__main__` check, that catches a `rospy.ROSInterruptException` exception, which can be thrown by `rospy.sleep()` and `rospy.Rate.sleep()` methods when Ctrl-C is pressed for instance.

**Subscriber** : As advanced in the previous paragraph ROS tutorials provide a basic implementation also of the subscriber node, that must be located, downloaded, and made executable as the publisher script:

```
wget https://raw.github.com/ros/ros_tutorials/kinetic-devel/
    rospy_tutorials/001_talker_listener/listener.py

chmod +x listener.py
```

The subscriber script should appear as follow:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # name is launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is
    stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

Listing 3.8: Python subscriber script

The code for the new node is fairly similar to the publisher one, except for the introduction of a callback mechanism for subscribing to the messages. In particular, the listener function declares a node which subscribes to the `chatter` topic which is of type `std_msgs.msgs.String` and when new messages are received, the callback is invoked with the message as the first argument. Finally, `rospy.spin()` keeps your node from exiting until the node has been shutdown. Before being able to execute the two scripts the `CMakeLists.txt` file should be updated to make sure the python script gets installed properly and uses the right python interpreter:

```
catkin_install_python(PROGRAMS scripts/talker.py scripts/listener.
    py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Next, the workspace must be rebuilt to notify ROS of the changes[72].

#### 3.6.4.0.1 Running the nodes

To test the nodes' behavior it is necessary firstly to run ROS master[13]:

```
roscore
```

Next, to source the workspace[14]:

```
source ./devel/setup.bash
```

Finally to run the two nodes in two different shells:

```
rosrun example_pkg talker.py
rosrun example_pkg listener.py
```

After having launched the commands one should see something similar to:

```
[INFO] [WallTime: 1314931831.774057] hello world 1314931831.77
[INFO] [WallTime: 1314931832.775497] hello world 1314931832.77
[INFO] [WallTime: 1314931833.778937] hello world 1314931833.78
[INFO] [WallTime: 1314931834.782059] hello world 1314931834.78
[INFO] [WallTime: 1314931835.784853] hello world 1314931835.78
[INFO] [WallTime: 1314931836.788106] hello world 1314931836.79
```

in the publisher terminal while in the subscriber shell the output should look like:

```
[INFO] [WallTime: 1314931969.258941] /listener_17657_13149319687951
     heard hello world 1314931969.26
[INFO] [WallTime: 1314931970.262246] /listener_17657_13149319687951
     heard hello world 1314931970.26
[INFO] [WallTime: 1314931971.266348] /listener_17657_13149319687951
     heard hello world 1314931971.26
[INFO] [WallTime: 1314931972.270429] /listener_17657_13149319687951
     heard hello world 1314931972.27
[INFO] [WallTime: 1314931973.274382] /listener_17657_13149319687951
     heard hello world 1314931973.27
[INFO] [WallTime: 1314931974.277694] /listener_17657_13149319687951
     heard hello world 1314931974.28
[INFO] [WallTime: 1314931975.283708] /listener_17657_13149319687951
     heard hello world 1314931975.28
```

To terminate the scripts as well as the ROS master it is necessary to press `Ctrl+C` in each terminal[71].

[13] For any problem related to ROS setup it is suggested to look at the section 7.4.

[14] It is important to remember that `catkin_make` command must be launched inside the workspace folder, in this case `example_ws`, if not it will fail.

# Chapter 4

# ROS Tools for Simulation

## 4.1 Background

In robotics, a simulator plays a key role, aiding the development of prototypes; indeed by emulating reality within a certain degree of rigor, it allows almost inexpensive and less consuming tests to develop architectures[10]. Those off-the-shelf components allow users to reduce the complexity of the robot management by skipping the complications of hardware development and setup, besides allowing online development and repetition.

### 4.1.1 Simulation in ROS

ROS simulators do not require complex operating procedures to be executed, as it is sufficient to spawn a roslaunch script, to wait a small amount of time to see the robot appear, and to push `Ctrl-C` to teardown the simulation.

Due to the isolation provided by the messaging interfaces of ROS, a vast majority of the robot's software graph can be run identically whether it is controlling a real robot or a simulated robot. At runtime, as the various nodes are launched, they find one another and connect; simulation input and output streams connect to the graph in the place of the device drivers of the real robot[43].

The following sections present the main tools used for ROS simulations.

## 4.2 Stage

### 4.2.1 Introduction

Stage is a two-dimensional simulator with a considerably simple modeling language that allows the creation of planar worlds with simple types of objects[40]. There are three ways to use Stage:

- using the *stage program*: a standalone robot simulation program that loads the robot control program from a chosen library.
- utilizing the *Stage plugin for Player* (`libstageplugin`) provides a large set of virtual robots for the popular Player networked robot interface system.
- writing a *personal simulator*.

### 4.2.2 Models

Stage provides a virtual world populated by mobile robots and sensors, along with various objects for the robots to sense and manipulate.

### 4.2.3 Design

Stage was designed from the outset to support multiple robots simultaneously interacting with the same world. It has been wrapped with a ROS integration package that accepts velocity commands from ROS and outputs an odometric transformation besides the simulated laser range-finders from the robot(s) in the simulation[43].

## 4.3 RViz

### 4.3.1 Introduction

RViz stands for *ROS Visualization*, it is the general-purpose 3D visualization environment for robots, sensors, and algorithms, whose main goal of the tool is to allow users to verify data by showing ROS messages in three-dimension. Therefore, the tool provides a graphical interface that allows user to visualize a lot of information about robot structure as well as the environment in which that latter is located; indeed, using RViz plugins the user can focus on a different aspect of the robot simulation as the model, or a particular sensor, for instance, analyzing how that latter perform during the testing.

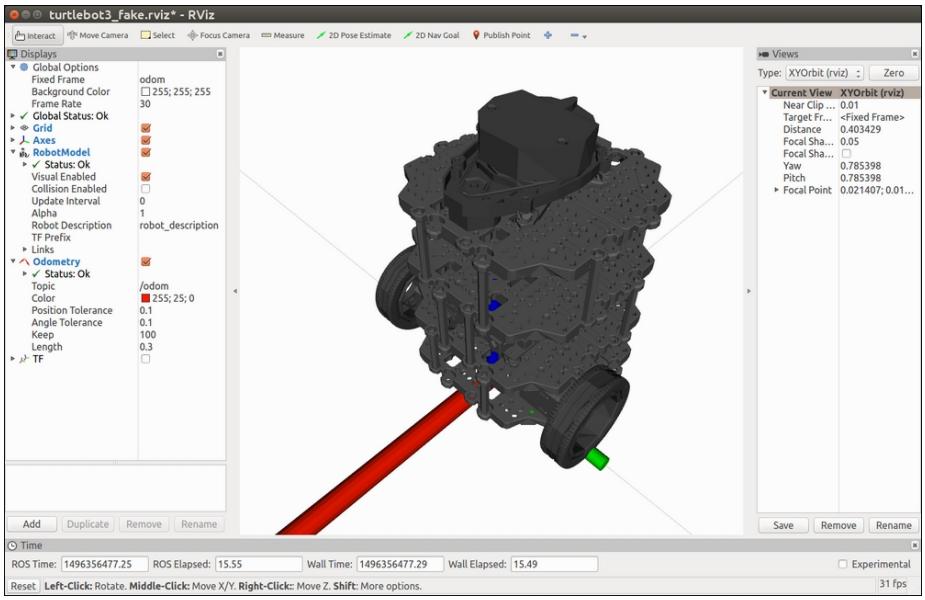


Figure 4.1: Turtlebot model simulation using RViz

### 4.3.2 GUI

RViz has numerous panels and plugins that can be configured as needed for a given task.

### 4.3.3 Screen Components

1. *3D View*: it is a black area at the center of the screen which primary screen allows users to see several data in three dimensions.

2. *Displays*: they are located on the left column, and they provide a way to select which data to display from the diverse topics.

3. *Menu*: The Menu bar is on the top of the screen; it contains some basic commands to save or load the current display settings, besides providing several panels for other operations.

4. *Tools*: below the menu bar there are the Tools, this panel furnishes a set of helpful functions such as interact, camera focus change, 2D navigation, etc.

5. *Views*: The Views panel sets up the viewpoint of the 3D view:

6. *Time*: Time shows the current time (wall time), ROS Time, and the elapsed time for each of them, moreover can be restart if needed.

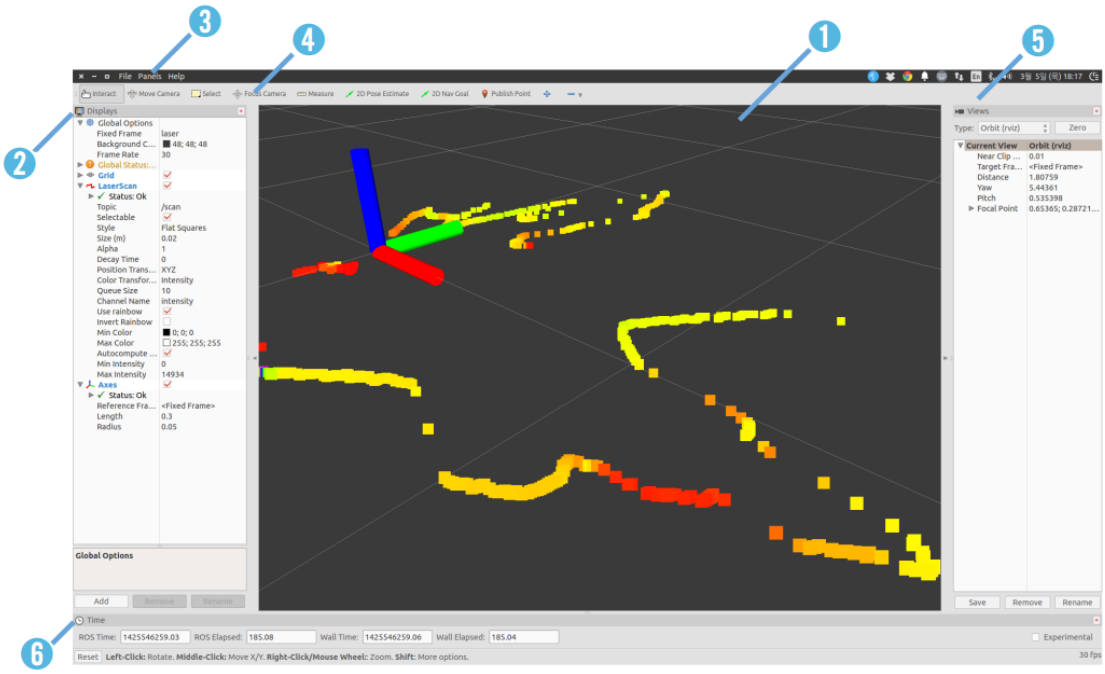


Figure 4.2: Rviz screen components'

#### 4.3.3.1 Display

Rviz display menu is used to select the message to display on the 3D View panel, and descriptions on each item are explained following tables[43]:

| Icon | Name | Description |
|---|---|---|
|  | Axes | Displays the xyz axes. |
|  | Camera | Creates a new rendering window from the camera perspective and overlays an image on top of it. |
|  | DepthCloud | Displays a point cloud based on the Depth Map. It displays distance values acquired from sensors such as Kinect and Xtion with DepthMap and ColorImage topics as points with overlayed color obtained from the camera. |
|  | Effort | Displays the force applied to each rotary joint of the robot. |
|  | FluidPressure | Displays the pressure of fluids, such as air or water. |
|  | Grid | Displays 2D or 3D grids. |
|  | Grid Cells | Displays each cells of the grid. It is mainly used to display obstacles in the costmap of the navigation |
|  | Group | This is a container for grouping displays. This allows us to manage the displays being used as one group. |
|  | Illuminance | Displays the illuminance. |
|  | Image | Displays the image in a new rendering window. Unlike the Camera display, it does not overlay the camera |
|  | InteractiveMarkers | Displays Interactive Markers. We can change the position (x, y, z) and rotation (roll, pitch, yaw) with the mouse. |
|  | LaserScan | Displays the laser scan value. |
|  | Map | Displays the occupancy map, used in navigation, on top of the ground plane. |
|  | Marker | Displays markers such as arrows, circles, triangles, rectangles, and cylinders provided by RViz. |
|  | MarkerArray | Displays multiple markers. |

Figure 4.3: RVIZ icon (first part)

| Icon | Name | Description |
|---|---|---|
|  | Odometry | Displays the odometry information in relation to the passage of time in the form of arrows. For example, as the robot moves, arrow markers are displayed showing the traveled path in a connected form according to the time intervals. |
|  | Path | Displays the path of the robot used in navigation. |
|  | Point Cloud | Displays point cloud data. This is used to display sensor data from depth cameras such as RealSense, Kinect, Xtion, etc. Since PointCloud2 is compatible with the latest Point Cloud Library (PCL), we can generally use PointCloud2. |
|  | Point Cloud2 | |
|  | PointStamped | Displays a rounded point. |
|  | Polygon | Displays a polygon outline. It is mainly used to simply display the outline of a robot on the 2D plane. |
|  | Pose | Displays the pose (location + orientation) on 3D. The pose is represented in the shape of an arrow where the origin of the arrow is the position(x, y, z,) and the direction of the arrow is the orientation (roll, pitch, yaw). For instance, pose can be represented with the position and orientation of the 3D robot model, while it can be represented with the goal point. |
|  | Pose Array | Displays multiple poses. |
|  | Range | This is used to visualize the measured range of a distance sensor such as an ultrasonic sensor or an infrared sensor in the form of a cone. |
|  | RelativeHumidity | Displays the relative humidity. |
|  | RobotModel | Displays the robot model. |
|  | TF | Displays the coordinate transformation TF used in ROS. It is displayed with the xyz axes much like the previously mentioned axes, but each axis expresses the hierarchy with an arrow according to the relative coordinates. |
|  | Temperature | Displays the temperature. |
|  | WrenchStamped | Displays the wrench, which is the torsion movement, in the form of 'arrow' (force) and 'arrow+circle' (torque). |

Figure 4.4: RVIZ icon (second part)

### 4.3.4 Usage

RViz tool is used for a wide range of different purposes, in particular, for what concerns hardware analysis; indeed, it provides a basic way to examine a robot's physical structure.

Nevertheless, due to the lack of a former education about robot modeling, it would be a complex task to analyze the tool usage in those terms, this is the reason why the tool is just used for navigation purposes in the use case section.

## 4.4 RQt

### 4.4.1 Introduction

RQt is a collection of more 30 GUI development tool which core elements were developed by Aaron Blasdel, from the University of Tokyo, at the Willow Garage[7]. In that name, the *r* is for ROS, and the *qt* refers to *Qt*[1]framework[34]. Thus RQt is a graphical user interface framework that implements various tools

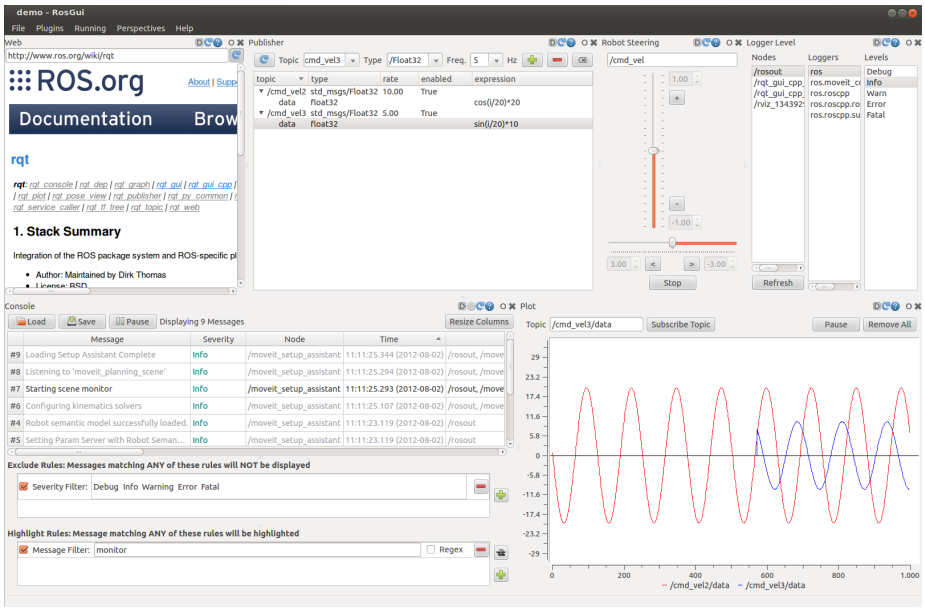


Figure 4.5: Rqt example of view interface

and interfaces in the form of plugins. One can run all the existing GUI tools as dockable windows within RQt! The tools can still run in a traditional standalone method, but RQt makes it easier to manage all the various windows in a single screen layout[45].

The tool's primary usage is nodes and topic visualization; indeed, using `rqt_graph` plugin, for instance, users can have a comprehensive view of the system on which they are working on. The tool creates a computation graph of the entire system that is used to visualize the nodes running on the system, and how they are communicating with each other.

In the next paragraph, RQt GUI toolkit will be examined.

[1]Qt: is a cross-platform framework used in the field of robotics and GUI development.[36][40]

### 4.4.2 Menu

RViz menu has four main sections:

- *File*: the File menu only contains the sub-menu to close 'RQt'.
- *Plugins*: there are over 30 plugins; organized in 9 sections: action, configuration, introspection, logging, miscellaneous tools, robot tools, services, topics, and visualization.
- *Running*: the currently running plugins are shown and they can be stopped when they are not needed.
- *Perspectives*: this menu saves operating plugins as a set and uses them later to run the same plugins.

### 4.4.3 Usage

RQt tool is a powerful tool used to analyze ROS systems; indeed, when large multi-node ROS systems are running understanding nodes' interactions can be a complex task. Nevertheless, RQt plugins provide users a simple way to analyze the system choosing the level of abstraction that they prefer. In this respect, RQt makes available a large set of devices that can be used to understand the system working process on one hand, and debugging it on the other.

This is the reason why, in the use case section, this tool will be widely used; especially that part that focuses on rqt_graph usage because that latter plugin is useful to examine the ROS system. In fact, by analyzing the graphs that it produces, one can easily understand several important information about the system on which one is working on. In particular, in the provided study case it will be used for debugging purposes since it will be examined only to control that all the nodes needed are running in the system and communicating on the right topics.

## 4.5 Gazebo

### 4.5.1 Introduction

Gazebo is a well-designed simulator, part of the Player Project[2], which allows simulation of robotic and sensors applications in three-dimensional indoor and outdoor environments making it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios[34]. In particular, the tool provides a simple way to visualize graphically a vast set of simulated environments that spread from basic houses and

[2]Player Project: is a project to create free software for research into robotics and sensor systems. Its components include the Player network server and the Stage robot platform simulators. [89]

gas stations to more specific ones, for instance, the Apollo landing site making robots able to interact with them. Indeed, Gazebo lets the user define the proprieties of each object of the simulation to improve user testing capabilities.

It has a Client/Server architecture and has a topic-based Publish/Subscribe model of inter-process communication[83].

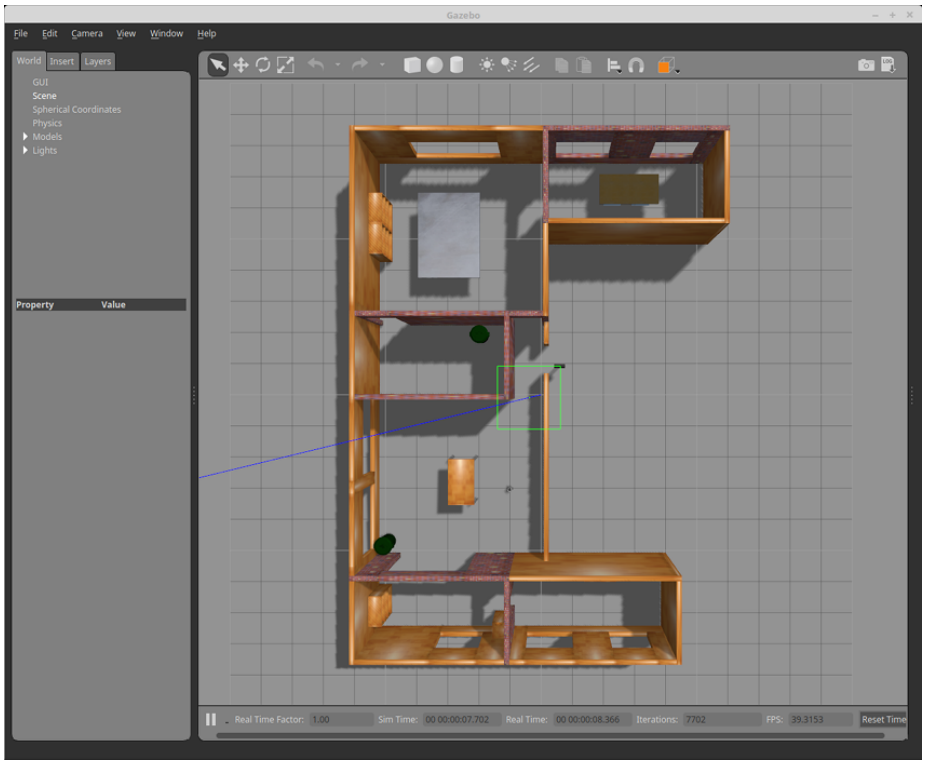


Figure 4.6: Gazebo empty house view

### 4.5.2 Features

Gazebo main features are:

- **Dynamics Simulation**: since version 3.0, besides ODE [3], various physical engines such as Bullet, Somebody, and DART are used by Gazebo to meet the needs of its users.
- **3D Graphics**: Gazebo uses OGRE[4], for both robot models along with the light, shadow, and texture to make them realistically drawn on the

[3] Open Dynamics Engine
[4] Open-source Graphics Rendering Engines

screen.

- **Sensors and Noise Simulation**: it supports a wide range of sensors (i.e Laser range finder[5], 2D/3D camera, contact sensor, etc.) and it makes it possible to apply noise to sensor data to make them as similar as possible to the actual environment.
- **Plugins**: it enables users to create robots, sensors, environment control as a plugin by providing APIs.
- **Robot Model**: it provides several robot model files, such as PR2, TurtleBot, and many others, moreover it lets users add their robots (if modeled as SDF file).
- **TCP/IP Data Transmission**: Gazebo allows users to run simulations on a remote server by using Google's Protobufs to pass socket-based messages.
- **Cloud Simulation**: Gazebo can be run on Amazon AWS by using CloudSim along with GzWeb to interact with the simulation through a browser.
- **Command Line Tool**: it uses GUI along with CUI tools to check and manage the simulation status.[40]

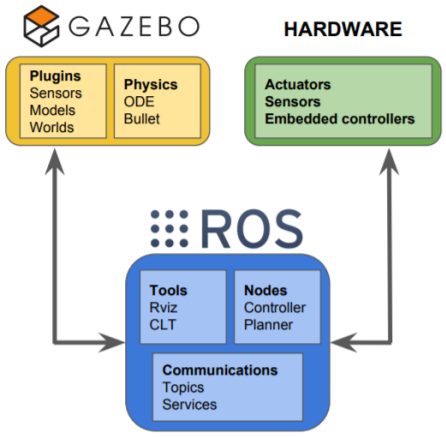


Figure 4.7: Gazebo interaction's graph

### 4.5.3 Architecture

Distributed architecture is used in Gazebo to decouple libraries for physics simulation, rendering, user interface, communication, and sensor generation.

Moreover, it provides two executable programs for running simulations:

[5] LRF

- a server `gzserver` for simulating the physics, rendering, and sensors;
- a client `gzclient` that provides a graphical interface to visualize and interact with the simulation

The client and server communicate using the gazebo communication library[21].

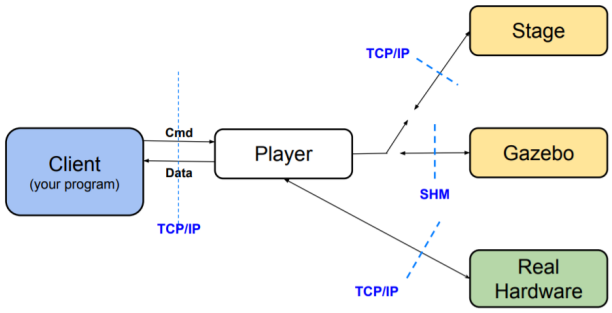


Figure 4.8: Gazebo communication's graph

#### 4.5.3.1 Client-Server

The `gazebo` command runs two different executables: the first is called *gzserver*, and the second *gzclient*. The `gzserver` is the core of Gazebo; it runs the physics update-loop and sensor data generation, besides it can be used independently of a graphical interface. The `gzclient` runs a *QT* based user interface which provides a visualization of simulation, and convenient controls over various simulation properties.

#### 4.5.3.2 Communication between processes

The communication library uses the open-source *Google Protobuf* for the message serialization and *boost::ASIO* for the transport procedure. It supports the publish-subscribe communication paradigm; that latter allows introspection of a running simulation, besides providing a convenient technique to control aspects of Gazebo.

The Gazebo clients can access its data through a shared memory; hence each simulation element in Gazebo can be associated with one, or more controllers that process commands for controlling the object and generate the state of that latter. Gazebo interfaces, called *Ifaces*, are used by controllers to publish the data they create into the shared memory, which can be read also by *Ifaces* of other processes to allow inter-process communication between the robot controlling software and Gazebo independently of the programming language or the computer hardware platform. The Client sends control data, as simulated objects' coordinates for instance, to the Server which draws up the real-time control of the simulated robot. Moreover, Gazebo allows users to perform a distributed simulation by placing the Client and the Server on different machines

using ROS Plugin for Gazebo to implement a direct communication interface to ROS and in that way controlling the simulated and the real robots using the same software. That provides an effective simulation tool for the testing and development of real robotic systems.[83]

#### 4.5.3.3 Gazebo Master

*Gazebo Master* is a topic name server which provides name lookup and topic management, besides a single master can handle multiple physics simulations, sensor generators, and GUIs.

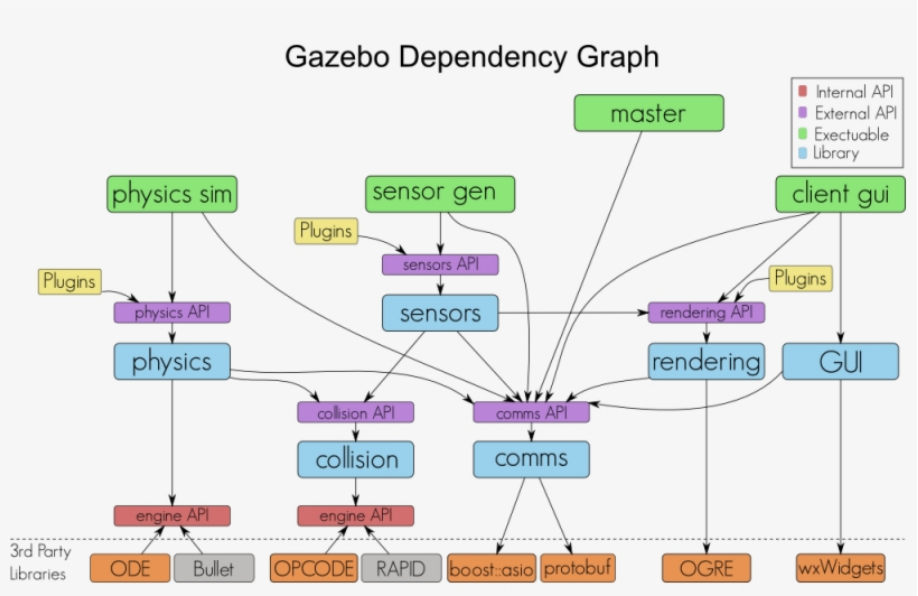


Figure 4.9: Gazebo dependency's graph

# Chapter 5

# Navigation Stack

## 5.1 Background

Packages in ROS are organized into ROS stacks. Packages' main goal is to create minimal collections of code for easy reuse, whereas stacks aim to facilitate the code sharing process. Especially, stacks are the primary mechanism in ROS for distributing software; each stack has an associated version and can declare dependencies on other stacks.

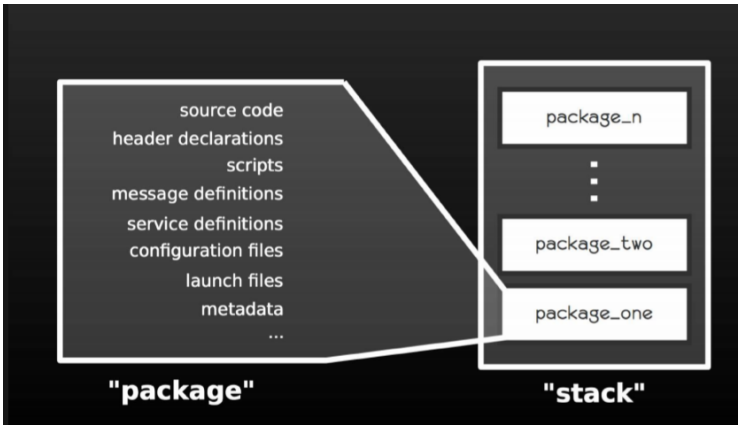


Figure 5.1: ROS stack and packages' main components

Stacks collect related packages from different communities that collectively provide functionality, such as a navigation stack or a manipulation stack. Unlike traditional software libraries stacks can also provide this functionality at runtime via ROS topics and services.

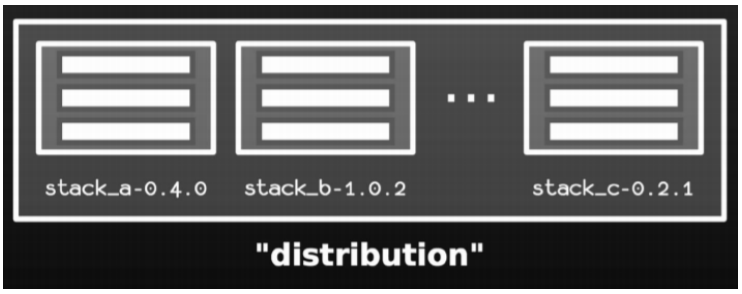


Figure 5.2: ROS distribution's building blocks

## 5.2 Introduction

ROS Navigation stack provides several helpful resources which enable a robot to navigate through a medium; namely, they make it be able to plan and following a path avoiding potential obstacles which may appear on its path throughout the course.

The primary objective of the package is to move a robot from the start position to the target position safely, which means without crashing or getting lost; to accomplish this task the package contains ready-to-use navigation algorithms which can be used in mobile robots, along with localization systems, which allow a robot to locate itself.

AMCL is the resource that enables the device to locate itself in an environment through a previously created map; the primary limitation of this tool is that allows a robot to navigate only in modification-free environments; as using a static map one would have to generate a new each time that the environment that surrounds suffered a modification. Luckily, the package offers two other localization systems to bypass the lack of flexibility of static maps both based on the SLAM technique: gmapping and hector mapping.

SLAM methods allow robots to map an environment during their navigation through it by using their sensors to gather information from the environment to generate a map. The main advantages of these approaches are that devices can generate a map of an unknown environment, on one hand, and updating an existing one on the other, overcoming in this way the previously presented limitation and supporting the use of robots in a more generic environment, not immune to changes. The two approaches differ essentially in the use of odometry information, as just the gmapping package takes it into account.[30]

To get odometry information, devices need to be equipped with special electromechanical sensors called encoders that are responsible for monitoring movement and position to provide position data from a robot[84]. This information enables the estimation of the robot's pose; therefore, more accurate maps can be generated. However, since not *not that glitters is gold*, odometry information to presents some limitation, in fact, the lack of precision on some factors' data as capitation, friction, slip, drift can accurate with time producing inconsistent

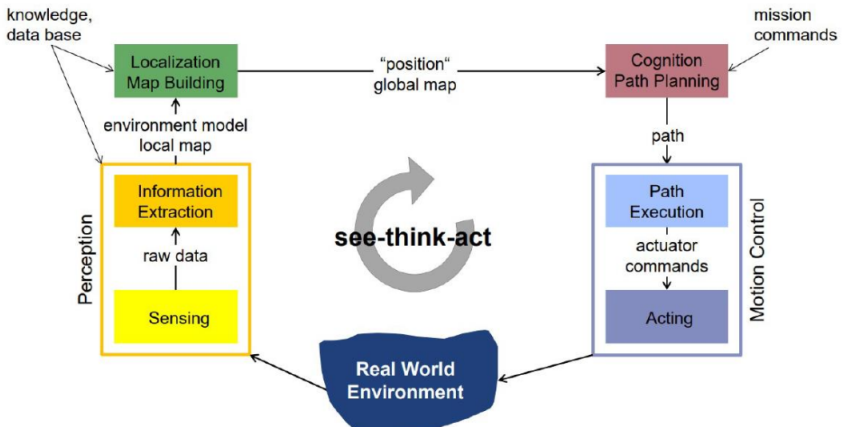


Figure 5.3: ROS Navigation's workflow

data that can lead to the generation of the distorted map. Encoders are not the only sensors need for the creation of a precise map, sensors' distance reading plays an essential role as well since they are in charge of the detection of the external world, which serves as a reference to the robot. Howbeit, since they measure the environment in relation to themselves, instead of in relation to the robot, the raw data need a geometric conversion to be used by the device to accomplish the navigation task. Luckily, ROS provides a tool also for this purpose, the TF package[1] [30]

## 5.3 Hardware Requirements

Nevertheless, the Navigation Stack is designed to be as general-purpose as possible, it has four basic limitations in respect to the hardware:

- it can manly handle *differential drive* and *holonomic wheeled robots*, although it is possible to use some features with another type of robots
- it assumes that the mobile base is controlled by sending desired velocity commands to achieve in the form of *twist message* type[2]: with X, Y and Theta velocities
- it needs to get environment information from a *LaserScan message* type topic[3]; therefore, it requires a planar laser to be mounted on the mobile base of the robot to build the map and perform localization.

[1] Information about TF package are at http://wiki.ros.org/tf

[2] Additional information about twist type message are at http://docs.ros.org/en/jade/api/geometry_msgs/html/msg/Twist.html

[3] Extra information about LaserScan message are at http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/LaserScan.html

- even though the Navigation Stack works on robots of arbitrary shapes and sizes, it performs better if they have nearly *square* or *circular shape* has it was developed on a square robot.[30]

## 5.4 Main components

In the following table the main components of the Navigation Stack will be presented along with a short description of their main purpose:

| Package/Component | Description |
|---|---|
| map_server | offers map data as a ROS Service |
| gmapping | provides laser-based SLAM |
| amcl | a probabilistic localization system |
| global_planner | implementation of a fast global planner for navigation |
| local_planner | implementations of the Trajectory Rollout and Dynamic Window approaches to local robot navigation |
| move_base | links together the global and local planner to accomplish the navigation task |

Figure 5.4: ROS Navigation Stack's main components

## 5.5 Planners

### 5.5.1 Global Planner

The global path planner in ROS operates on the `global_costmap`, which generally initialized from a *prior static map*, then it could be updated frequently based on the value of `update_frequency` parameter. The global path planner is responsible for generating a long-term plan from the start or current position to the goal position before the robot starts moving. It will be seeded with the costmap, as well as, the start and goal positions. Those start and goal positions are expressed by their `x` and `y` coordinates. A grid-based global planner that can use *Dijkstra*'s algorithm or *A** algorithm to compute shortest collision free path for a robot is obtained in `global_planner` package.

Also, ROS provides another global planner named `arrot_planner`, which is a simple planner that attempts to move the robot as close to its goal as possible even when that goal is in an obstacle.

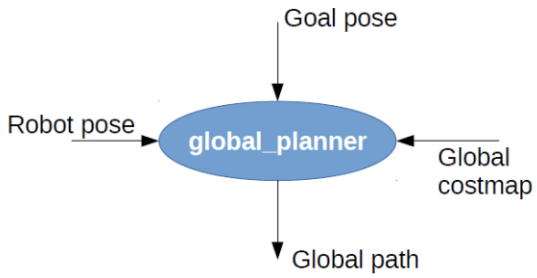


Figure 5.5: ROS global planner's diagram

### 5.5.2 Local Planner

The local path planner or the controller in ROS operates on the `local_costmap`, which only uses local sensor information to build an `obstacle map` and dynamically updated with sensor data.

It takes the generated plan from the global planner, and it will try to follow it as close as possible considering the kinematics and dynamics of the robot as well as any moving obstacles information in the `local_costmap`. ROS provides

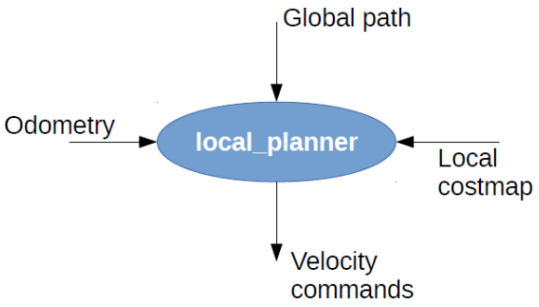


Figure 5.6: ROS local planner's diagram

implementation of two local path planning algorithms: namely the *Trajectory Rollout* and the *DWA*[4] in the package `base_local_planner`.
Both algorithms have the same idea to first discretely sampled the control space than to perform forward simulation, and the selection among potential commands. The two algorithms differ in how they sample the robot's control space.

[4]DWA: Dynamic-Window Approach

### 5.5.3 Local and Global Costmaps

The local and global 2D *costmaps* are the topics containing the information that represents the projection of the obstacles in a 2D plane (the floor), as well as a security inflation radius, an area around the obstacles that guarantee that the robot will not collide with any objects, no matter what is its orientation. These projections are associated with a cost, and the robot's objective is to achieve the navigation goal by creating a path with the least possible cost. While the global costmap represents the whole environment (or a huge portion of it), the local costmap is, in general, a scrolling window that moves in the global costmap in relation to the robot's current position.

## 5.6 Localization Systems

One of the resources needed to accomplish the Navigation task is the localization system that allows a robot to locate itself, whether there is a static map available or simultaneous localization and mapping is required.

### 5.6.1 AMCL and Map_server

AMCL and `map_server` are optional blocks, responsible for the static map usage.

#### 5.6.1.1 Map_server

**map_server** contains two nodes: `map_server` and `map_saver`. The first one, namesake to the package, as the name indicates, is a ROS node that provides static map data as a ROS Service, while the second one, `map_saver`, saves a dynamically generated map to a file.

### 5.6.2 SLAM

#### 5.6.2.1 AMCL

AMCL does not manage the maps, it is a localization system that uses the `base_footprint` or `base_link` transformation to the map to work, therefore it needs a static map, and it will only work after a map is created. This localization system is based on the *Monte Carlo localization* (MCL) approach: it randomly distributes the particles in a known map, representing the possible robot locations, and then uses a particle filter to determine the actual robot pose.

#### 5.6.2.2 gmapping

Gmapping, as well as AMCL, is a localization system, but unlike AMCL, since it works over the odom to map transformation, it does not need a map or IMU information, in fact, it just needs odometry data to perform *SLAM*. It

creates a 2D occupancy grid map using the robot pose and the laser data (or converted data).

#### 5.6.2.3 Hector_mapping

`hector_mapping` can be used instead of using gmapping. It uses the **base_link** to map transformation and, it does not need the odometry nor the static map: it just uses the laser scan and the IMU information to localize itself.

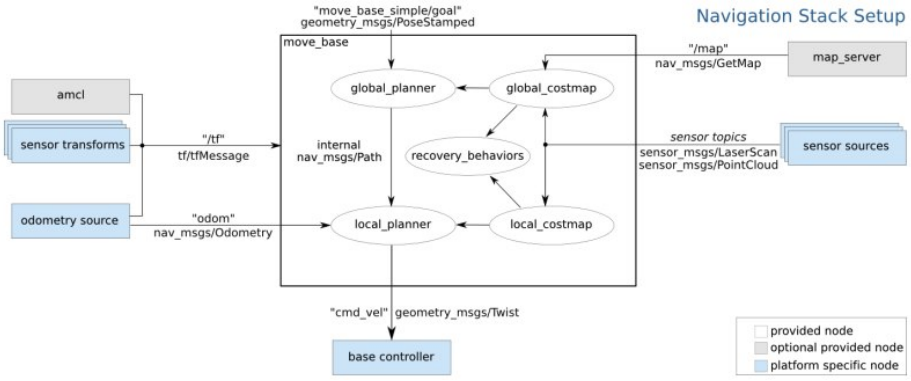


Figure 5.7: ROS Navigation Stack setup's graph

## 5.7 Navigation Types

Our robot will move through the map using two types of navigation: global and local.

### 5.7.1 Global Navigation

Global navigation is used to create paths for a goal in the map or a far-off distance; it is performed by using *Dijkstra* e *A**.

### 5.7.2 Local navigation

Local navigation is used to create paths in nearby distances and avoid obstacles; it uses *DWA*[5] e Trajectory Rollout.

[5]DWA: Dynamic-Window Approach

## 5.8 Frames

In this section will be illustrated the basic frames of Navigation Stack, in fact in order to work properly, the robot needs to publish the following tf relationships: center of the robot(base link). `/map →/odom →/base_link →/base_laser`

- **map** is the global reference frame and the robot pose should not drift much over time in relation to it.
- **odom** drifts and can cause discrete jumps when new sensor information is available.
- **base_link** is attached to the robot center.
- **base_footprint** is very straightforward: it is the `base_link` projection on the ground (zero height). Therefore, its transform is published in relation to the `base_link`.
- **base_stabilized** is the center position of the robot, not computing the roll and pitch angles. Therefore, its transform is also published in relation to the `base_link`.
- **laser_link** is the center position of the laser sensor, and its transform is published in relation to the `base_link`.

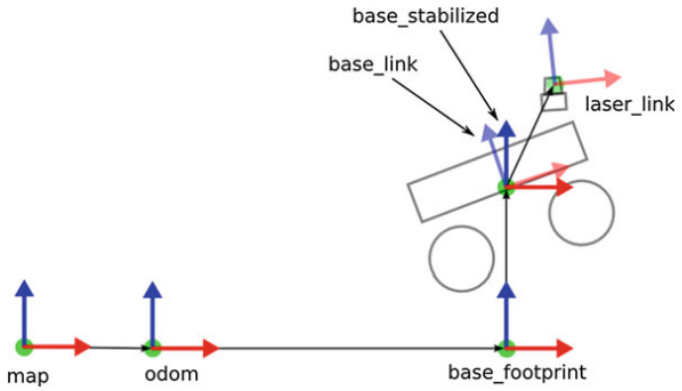


Figure 5.8: Potential frames of interest in a 2D view of a robot travelling through rough terrain

## 5.9 move_base node

The `move_base` package, is the major component of the Navigation Stack which primary goal is to displace a robot from its actual position to the target

one.

One of the nodes of the package is the move_base node. IT links the global-planner and the local-planner for the path planning, it connects to the rotate-recovery package if the robot is stuck in some obstacle, and it connects the global costmap and local costmap for getting the map.

The move_base node is an implementation of Action Server10.2 which takes a goal pose with message type (`geometry_msgs/PoseStamped`) that can be sent using a Action Client node.

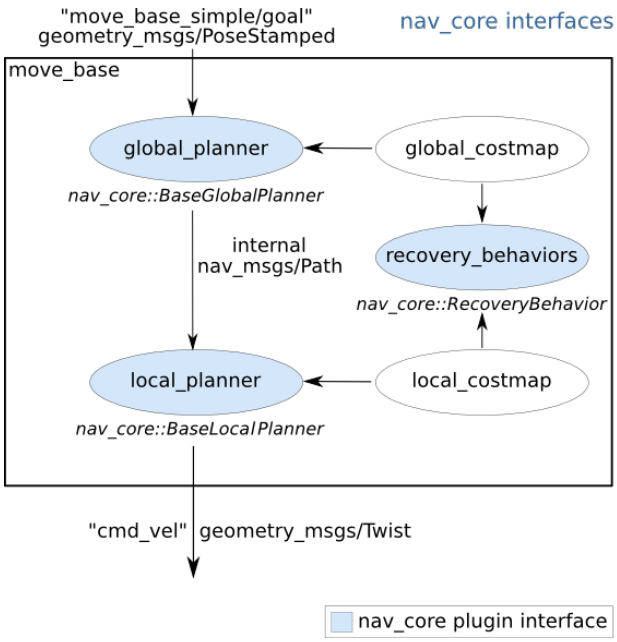


Figure 5.9: ROS move_base graph about how ROS topic are used to accomplish navigation goal

The `move_base` node subscribes to the `move_base_simple/goal` topic, which is the input of the Navigation stack. When this node receives a goal pose, it links to elements such as `global_planner`, `local_planner`, `recovery_behavior`, `global_costmap`, and `local_costmap`, generates the output which is the command velocity (`geometry_msgs/Twist`), and sends to the base controller for moving the robot for achieving the goal pose.

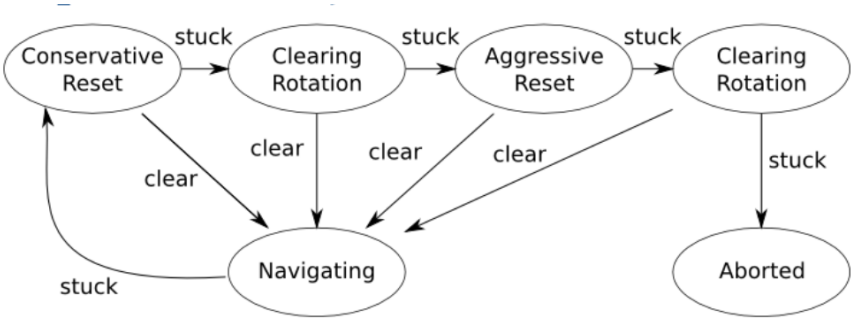


Figure 5.10: ROS move-base recovery behaviour

# Chapter 6

# Turtlebot

## 6.1 Introduction

TurtleBot is a ROS low-cost, personal robot kit with open-source software which can be used for a wide range of purpose, which spread from educational and research use to hobbies and product prototyping.

## 6.2 History

TurtleBot is a ROS standard platform robot created by Tully Foote and Melonee Wise at Willow Garage on top of the iRobot's Roomba-based research robot, Create, for ROS deployment in 2010.

### 6.2.1 Why a turtle?

Turtlebot is originated from the Turtle robot, which was driven by Logo (10.2). Moreover, the *turtlesim node*[1], which first appears in the basic tutorial of ROS, is a program that mimics the command system of the *Logo turtle program* 10.2. It is also used to create the Turtle icon as a symbol of ROS. The nine dots used in the ROS logo are derived from the back shell of the turtle. TurtleBot, which originated from the Turtle of Logo, is designed to easily teach people who are new to ROS through TurtleBot besides teaching computer programming language using Logo. Since then TurtleBot has become the standard platform of ROS, which is the most popular platform among developers and students.[15]

[1] turtlesim node:http://wiki.ros.org/turtlesim


Figure 6.1: TurtleBot icon

## 6.3 TurtleBot3

There are three versions of the TurtleBot model: Turtlebot1, Turtlebot2, and Turtlebot3; created respectively in 2010, 2012, and 2017. Since the later version is the one that provides the larger set of functionalities, it is one chosen to develop the case uses that will be presented in the next part. In the following sections will be given a brief overview about Turtlebot3 main characteristics.

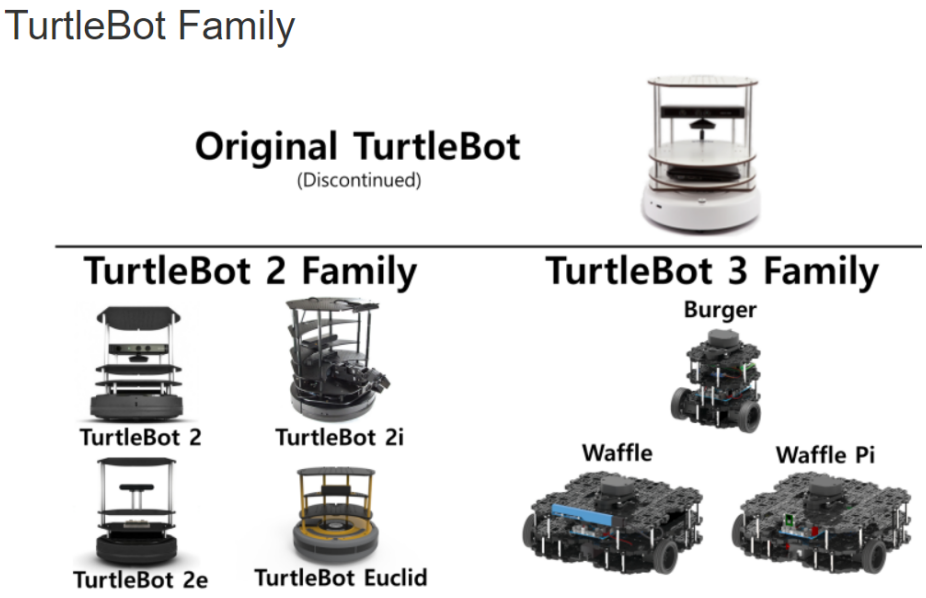


Figure 6.2: Turtlebot

### 6.3.1 Hardware description

TurtleBot3 is a small, affordable, programmable, ROS-based mobile robot developed with features to overcome the lacking functions of its predecessors and the demands of users. The primary objectives during the development of the latter Turtlebot version were on one hand, to decrease the size of the robot and, on the other to reduce its price without renounce to quality and to offer expandability.

It is sold in two main configurations, referred to as the "burger" and "waffle" models; the second one is the more expensive as it is slightly larger, besides having an additional camera sensor.

The main hardware components of the platform are:

- *LIDAR*: a 360°laser scanner is mounted on top of the robot. This sensor transmits a rotating laser, which reflects nearby obstacles. By using the reflection time, distances to nearby objects can be obtained, which can be used for mapping or obstacle avoidance.
- *Raspberry Pi*: a Raspberry Pi 3 single-board computer is set one level below the LIDAR. It provides the master (a pc) the data that it reads from the sensors, letting it elaborate the computation.
- *OpenCR*: one level below the Raspberry, is located the hardware control board, which is based around an Arduino Uno. The board includes an Inertial Measurement Unit (IMU) with a three-axis accelerometer, gyroscope, and magnetometer. It provides the connection from the Raspberry Pi to the motors and sensors besides furnishing power connections for all elements.
- *motor* and *batteries* are located on the lowest level. Both Turtlebot3 models have a differential drive configuration[2] [2]

## 6.4 Case Study usage

Due to the lack of a Turtlebot robot on the campus of Cesena, besides the impossibility to reach an external laboratory because of the Coronavirus emergency, the use case section only uses a Turtlebot3 simulator. That latter is an accurate model of the real turtlebot hardware platform, indeed, it represents a simple way to test how to perform turtlebot tasks, like Navigation, exploiting its sensor and actuators. Before continuing, it is valuable to underline that the following examples will not analyze Turtlebot sensors and actuators' usage, they will use high-level algorithms without focusing on how they handle Turtlebot hardware.

[2]Differential drive: having two independently driven motors and one caster wheel for stability

# Chapter 7

# Use cases

## 7.1 Background

This chapter assumes that the lector has some Linux and Unix-based systems environment literacy, for that reason instructions about general Linux environment configuration are not given.

Since I had no formal education about ROS before writing this document and ROS has a vast set of distributions as well as robot platforms, I decided to base my preliminary decisions about the target robot as well as ROS distribution to use mainly on the indications provided in the slides[35] of the course of *Industrial Robotics* of the University of Bologna.

## 7.2 Introduction

This first chapter of the use case section is about the preliminary pc initialization, especially ROS chosen distribution, along with the relative Linux version and robot objective platform will be identified. Furthermore, this part also provides some information about ROS environment variables' setup besides some suggestions to control if these latter are initialized correctly. Lastly, the chapter ends with a description of some common ROS commands which will be extensively in the subsequent parts.

## 7.3 Requirements

This section gives a list of requirements to run the following examples.

### 7.3.1 ROS distribution

The ROS distribution chosen to implement the project is ROS *Kinetic Kame* distribution[57]; as an advance in the paragraph 7.1, that version had been

chosen because it suggested in University courses such as the one of *Industrial Robotics* of the University of Bologna[35].


Figure 7.1: ROS Kinetic Kame distribution icon

Still in that paragraph, some assumptions about user Linux knowledge were made. In reason of them, instructions on how to install ROS Kinetic distribution along with ROS official build system[1] will not be provided, however, one can find all the information needed for the purpose at the following links: http://wiki.ros.org/kinetic/Installation/Ubuntu and http://wiki.ros.org/catkin.

### 7.3.2 Linux version

ROS Kinetic distribution is primarily available for Ubuntu Wily (15.10) and Ubuntu Xenial (16.04 LTS)[2], moreover it presents some unofficial and experimental platform as well.

### 7.3.3 Robot Hardware

Since the Turtlebot is run just as a simulation, there is no hardware requirement. However, the next sections give some information about how to set up the simulated robot in a ROS kinetic-based environment.

## 7.4 Environment presetting

Before starting to program in ROS, its environment variables need to be initialized; in particular, the following code needs to be added to the `bash.rc` script. This operation is needed to be able to run the examples without having to run the following code in each new terminal.

```
export ROS_ROOT=/home/user/ros/ros

export PATH=$ROS_ROOT/bin:$PATH
```

[1] Catkin is ROS official build system since *Groovy* distribution; more details about it can be found in section 10.2

[2] Ubuntu releases are at the following link: https://releases.ubuntu.com/

```
export ROS_MASTER_URI=http://{IP_ADDRESS_PC}:11311/

export ROS_HOSTNAME={IP_ADDRESS_CURRENT_PC}

export PYTHONPATH=$PYTHONPATH:$ROS_ROOT/core/roslib/src

echo "export TURTLEBOT3_MODEL=burger"
```

Listing 7.1: Commands to setup ROS environments variables

The first and second commands, `ROS_ROOT` and `PATH` enable the system to find the installed ROS core packages, the third `ROS_MASTER_URI` provides nodes information about where the master can be found, it should be set to the *XML-RPC URI* of the master.One might just choose to use `localhost` as a value for this variable, however, it is not recommended because it could lead to unintended behaviors with remotely launched nodes.

Moreover, the `PYTHONPATH` variable needs to be updated, whether one decides to program in Python or not because many ROS infrastructure tools rely on Python and need access to the `roslib` package for bootstrapping[53].

Finally, the last command is the one that specifies to the system the target model of turtlebot used.

## 7.5 Environment control

To execute the use case of the next chapters ROS environment must be initialized correctly; to check if the environment variables had been initialized correctly it is profitable to run the following commands.

1. checks if ROS variables as ROS_ROOT and ROS_PACKAGE_PATH are set by launching:

   ```
   printenv | grep ROS
   ```

   If one detect that they are not set, then he should source some set.*sh files.

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

   Given that this command must be run on every new shell it is easier to add it to `/.bashrc` file because it is run automatically in every new terminal window. Moreover, this operation allows users to install several ROS distributions on the same computer and switch between them by sourcing the distribution-specific setup.bash file.

**Troubleshooting** If the environment variables have not been sourced correctly; one could have a `roscore: command not found`; if that happens is because the environment is not set up correctly; therefore, it could be useful to take a step back and refer to the previous section or at https://wiki.ros.org/ROS/NetworkSetup#Single_machine_configuration for network-related issues[73].

## 7.6 Useful commands

### 7.6.1 roscore

The first command to be launched is `roscore`, that latter is one of the most significant commands in the ROS system since it is a collection of programs that are pre-requisites of a ROS-based system, thus, nodes are not able to communicate if it is not running properly.

There are several components that are started with roscore:

- a ROS Master (3.3.2)
- a ROS Parameter Server (3.3.2.1)
- a rosout logging node

It is valuable to remember that ROS master should continue running for the entire time that one is using ROS; as a matter of fact, one reasonable workflow is to start `roscore` in one shell, then open other terminals for the "real" work. The most general reason to stop `roscore` is to have finished working with ROS. Once that point is reached, the master can be stopped by using the *Ctrl+C* technique, however, that method may not give the node a chance to unregister itself from the master. As a consequence of this issue, one could still see the killed node launching `rosnode list` for a while; to avoid this uncomfortable situation one can execute the `rosnode cleanup` command to remove dead nodes from the list. The `roscore` command must be executed only once because ROS does not allow multiple masters in the system[75]. Next, the `rosrun` command, is ROS basic command to create a node. It requires two parameters: a package name and the name of an executable file within that package.

### 7.6.2 rosrun

`rosrun` command is only a shell script that can understand the organization of ROS files finding the executables required by their package names, and then executing them; avoiding users to specify the complete path of the files. The work of registering with the master to become a ROS node happens inside the program, not in `rosrun`.

### 7.6.3 roslaunch

`roslaunch` is a tool that enables multiple ROS nodes to be launched locally or remotely via SSH; furthermore, it provides a way to initialize parameters on the Parameter Server. Options to automatically respawn processes that have already died are contained in `roslaunch` too. [62] Especially, `roslaunch` takes in one or more XML configuration files with the `.launch` extension that specifies the parameters to initialize and nodes to execute, besides the machines on which they should run. Indeed `roslaunch` command can evaluate an XML file in a single pass as by analyzing them in a depth-first traversal order: tags are

examined sequentially, and the last configuration is the one that it is applied. Nevertheless, since there is no guarantee that an override is specified correctly relying on the override behavior can be brittle, thus, it is better to implement that mechanism using $(arg)/¡arg¿ settings.

## 7.7 Project package

There are several packages and tutorials to use turtlebot, the primary purpose of this one is not to provide a complete turtlebot package to handle all the tasks that they can perform, on the contrary, it aims to collect a minimal set of files to implement turtlebot navigation in a given context.

The repository contains a workspace with three packages: two of them, `gazebo_models` and `turtlebot3_ds` are used in turtlebot examples, while the last one, `turtlesim_cleaner`, for `turtlesim` simulation. `gazebo_models` In particular, `gazebo_models` only provides the model files of some Gazebo worlds to be able to visualize them correctly in the simulation. `turtlebot3_ds`, on the other hand, holds three main directories: one to perform robot simulation, one to enable user control a multiple turtlebot navigation, and the latter stores' other files need by Gazebo. In particular, `turtlebot3_description_ds`, contains the turtlebot model files required to visualize the robot in Gazebo[3], the `turtlebot3_navigation_ds` holds the launch files to execute the simulation and finally the `worlds` directory stores the file that defined the properties of the worlds used for the simulation.

[3]Since there was no point in trying to re-implement turtlebot model that latter has been imported from https://github.com/ROBOTIS-GIT/turtlebot3/tree/master/turtlebot3_description

# Chapter 8

# ROS case use

## 8.1 Introduction

This chapter is about some simple use cases to understand ROS core elements. Since ROS has a vast ecosystem which dimensions boost day by day it is not possible to examine all of them, that is the reason why this section focuses only on two of the core building blocks using the first example of simulation in the ROS environment to analyze them. Given that LOGO heritage had a considerable relevance in ROS development, the initial example is implemented using a turtle avatar. Specifically, in the first place, the turtlesim 8.2 simulator will be used to understand how ROS nodes and topics work.

Next, in the latter part of this section, it is shown an initial facile example of how ROS can be run across multiple machines aside from a detailed digression focused on the publish-subscribe pattern consequence in the ROS ecosystem.

## 8.2 Understanding ROS Nodes

As advanced in the previous paragraph the following examples aim to understand ROS nodes and topics using the *turtlesim simulator*, especially, since this first section is centered on nodes. ROS *turtlesim simulator* is a lightweight simulator for learning ROS, besides giving an idea of what the later robot simulations will be about. An overview of ROS `turtlesim` package can be seen in the following image:

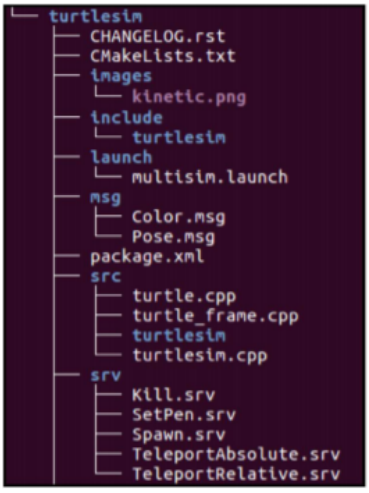


Figure 8.1: ROS turtlesim package

To implement the case use the first step to take is to launch `roscore` command in a shell and `turtlesim_node` in another, indeed that expedient (running nodes in different terminals) is necessary to execute simultaneously commands. That former node, `turtlesim_node`, provides a simple simulator for teaching ROS concepts; notably, it is responsible for the creation of the screen image and the turtle[1][4]. To do this one should use `rosrun` command as shown in the following example:

```
roscore
rosrun turtlesim turtlesim_node
```

By executing the second command one should see a graphical window opening; this latter window shows a simulated, turtle-shaped robot that lives in a square world [2].

Right now by running `rosnode list` command one should get the following output:

```
/rosout
/turtlesim
```

which means that two nodes are currently running in ROS, rosout and the turtlesim node. The first one is always running when programming on ROS because it is responsible for the collection and logging of the debugging output of nodes [74]. On the other hand, the second node is an instance of an executable called turtlesim_node responsible for creating the turtlesim window and simulating the turtle motion.

[1]The turtle may change from the one shown[74]

[2]The appearance of the turtle may be different every time the command is executed since the simulator randomly choose a turtle among a collection of "mascot" turtles for each of the historical distributions of ROS[34]

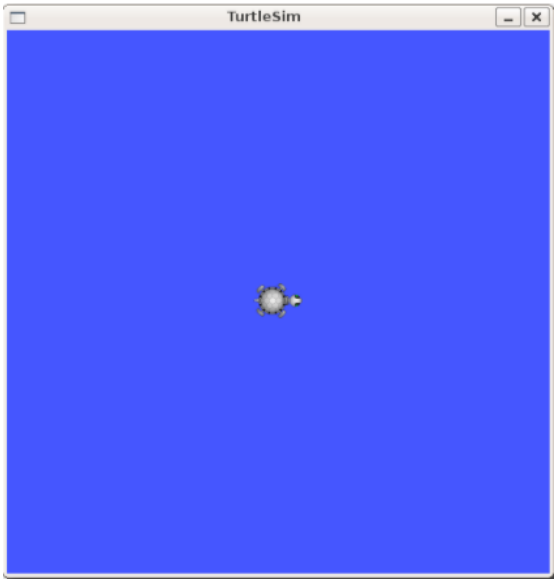


Figure 8.2: ROS turtlesim's avatar image

Now, if one teardown turtlesim node (pressing *Ctrl+C*), one can re-run the former command with a remapping argument[3]

```
rosrun turtlesim turtlesim_node __name:=my_turtle
```

Now by re-running the `rosnode list` command one should see that a new node,called `my_turtle`, took the place of the `turtlesim` one. To check if `my_turtle` node is up one could try to ping[4] it by launching:

```
rosnode ping my_turtle
```

which output should be something like:

```
rosnode: node is [/my_turtle]
pinging /my_turtle with a timeout of 3.0s
xmlrpc reply from http://aqy:42235/     time=1.152992ms
xmlrpc reply from http://aqy:42235/     time=1.120090ms
xmlrpc reply from http://aqy:42235/     time=1.700878ms
xmlrpc reply from http://aqy:42235/     time=1.127958ms
```

[3]Remapping arguments is a powerful feature of ROS that allow users to launch the same node under multiple configurations by the shell, besides remapping function is needed also because the ROS master does not allow multiple nodes with the same name. Further information can be retrieve at: http://wiki.ros.org/Remapping%20Arguments

[4]Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network. It is available for virtually all operating systems that have networking capability, including most embedded network administration software[88].

### 8.2.1 Topics

In this part besides `roscore` and `turtlesim_node` will be run a third node that provides a proper way to guide the prior little turtle avatar. This last node is called `turtle_teleop_key` and allows users to control the turtle to motion using the arrow keys of the keyboard.

```
roscore
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
```

The new node launched is an instance of an executable called `turtle_teleop_key`[5] This node waits for an arrow key to be pressed, converts that keypress to a movement command, and sends that command to the `turtlesim_node` node[6].

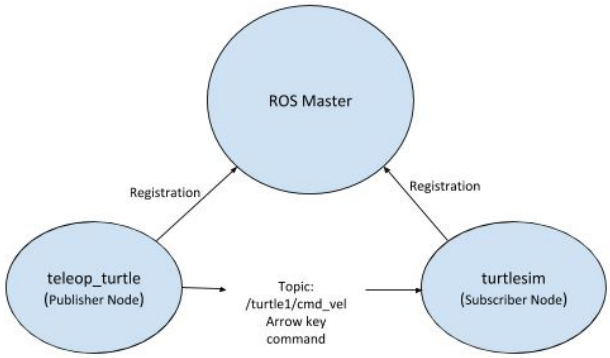


Figure 8.3: ROS: turtlesim communication's graph

#### 8.2.1.1 Nodes communication

As advanced in the paragraph 3.3.2, ROS nodes can communicate using the Publish-Subscribe pattern (10.2), which in turn use ROS topics. Notably, on one side `turtle_teleop_key` node publish keystrokes on a topic, while on the other side `turtlesim` node subscribes to the same topic to receive them.

One can visualize which nodes are communicating and on which topic they are doing it by executing the `rqt_graph` command which can build graph of the nodes of the system as shown in the figure 8.4. In the previous graph, the ovals represent nodes, while the directed edges represent publisher-subscriber relationships; indeed the picture shows that the node named `/teleop_turtle` publishes messages on a topic called `/turtle1/cmd_vel`[7], and that the node named `/turtlesim` subscribes to those messages.

[5]The abbreviation "teleop" stands for "teleoperation", which refers to to situations in which a human controls a robot remotely by giving direct movement commands[34].

[6]If the turtle does not move in response to the key presses, one should check if the turtle_teleop_key window has the input focus, for example by clicking inside it.[34]

[7]In this context, the name cmd_vel stands for "command velocity"[34]


Figure 8.4: ROS turtlesim graph when running turtle_teleop_key node

However, although the `rosout` node is still running it is missing from the graph's view, this is a direct consequence of `rqt_graph` standard settings, in fact, it usually hides by default nodes whose only reason for existence is debugging to avoid users unnecessary noises and letting them focus on the application developing. To disable that feature it is necessary to *uncheck* the *Hide debug* box; once that operation is done one should see that all of the nodes publish messages on a topic called `/rosout`, to which the node named `/rosout` subscribes, as shown in the next diagram.

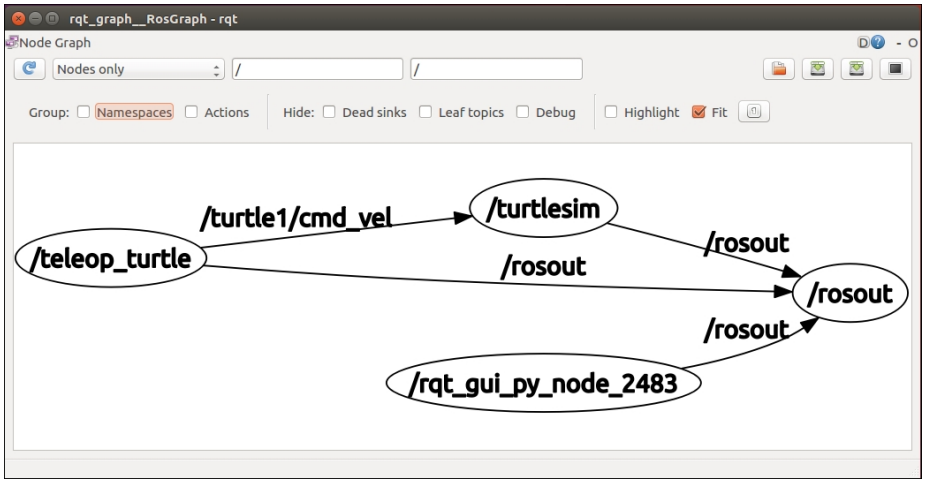


Figure 8.5: ROS rqt_graph of the whole turtlesim example

As exposed in section 3.3.4, to reduce the level of coupling between individual nodes, ROS nodes are designed to publish information that they have without worrying about whether anyone is subscribing to those messages. Indeed, when a key is pressed, the `/teleop_turtle` node publishes messages with those movement commands on a topic called `/turtle1/cmd_vel` and given that the turtlesim_node is subscribed to the same topic it receives those messages

simulating the turtle moving with the requested velocity.

This latter element allows user to get information about ROS topics, in particular, if it is run with the "-h" option it shows a list of the command related to rostopic:

```
rostopic -h
```

### 8.2.2 Creating a motion node

Using built-nodes is not the only way to control a simulator, or a robot in general, one could also want to implement one's algorithm to perform a particular task. In this section is shown a basic `Python` script to move the turtlesim simulator on a straight line. In particular, the script provides a simple way to control the avatar's velocity, direction, and distance it must travel. ROS tutorials provide the script, nevertheless, to avoid the user creating a new package and update the related dependencies in the `CMakeList.txt` file, a `turtlesim_cleaner` package containing the script has been created in the project repository.

**Motion script** Before analyzing the code, it is important to remember to make it executable by running the following command in the script folder:

```
  chmod u+x move.py
```

Next, it is useful to understand how

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist

def move():
    # Starts a new node
    rospy.init_node('robot_cleaner', anonymous=True)
    velocity_publisher = rospy.Publisher('/turtle1/cmd_vel', Twist,
     queue_size=10)
    vel_msg = Twist()

    #Receiveing the user's input
    print("Let's move your robot")
    speed = input("Input your speed:")
    distance = input("Type your distance:")
    isForward = input("Foward?: ")#True or False

    #Checking if the movement is forward or backwards
    if(isForward):
        vel_msg.linear.x = abs(speed)
    else:
        vel_msg.linear.x = -abs(speed)
    #Since we are moving just in x-axis
    vel_msg.linear.y = 0
    vel_msg.linear.z = 0
    vel_msg.angular.x = 0
    vel_msg.angular.y = 0
```

```
    vel_msg.angular.z = 0

    while not rospy.is_shutdown():

        #Setting the current time for distance calculus
        t0 = rospy.Time.now().to_sec()
        current_distance = 0

        #Loop to move the turtle in an specified distance
        while(current_distance < distance):
            #Publish the velocity
            velocity_publisher.publish(vel_msg)
            #Takes actual time to velocity calculus
            t1=rospy.Time.now().to_sec()
            #Calculates distancePoseStamped
            current_distance= speed*(t1-t0)
        #After the loop, stops the robot
        vel_msg.linear.x = 0
        #Force the robot to stop
        velocity_publisher.publish(vel_msg)

if __name__ == '__main__':
    try:
        #Testing our function
        move()
    except rospy.ROSInterruptException: pass
```

Listing 8.1: Turtlesim move forward script

As can be notice the script is similar to the ones present in the section 3.6.4, indeed the script creates a new node called `robot_cleaner` and a publisher on `/turtle1/cmd_vel` topic[8], namely, the topic used by turtlesim node to move around. Next, it creates and initializes the new message that will be sent to the turtlesim node asking the user the direction, velocity, and distance the simulator should travel. After this initialization procedure the script loops begin: that latter basically continues to publish the user desired velocity until the robot has travel the chosen distance, at that point the velocity is set to zero to stop the robot motion. As can be seen, the ROS custom node, `robot_cleaner` node, is

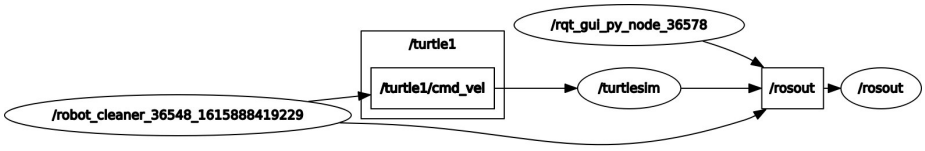


Figure 8.6: ROS rqt_graph when running turtlesim node and robot_cleaner node

visualized as any other node in the ROS system, furthermore: it is valuable to

[8] `cmd_vel` is the topic on which the motion messages are published: in particular the most common kind of messages used for the purpose are the Twist messages[48]. Those latter express velocity in free space broken into its linear and angular parts.

notice that number that follows the name of the node are the one created to make it unique in the ROS system because the value of the `anonymous` is `true`.

### 8.2.3 Running ROS across multiple machines

Given that ROS is designed with distributed computing in mind, deploying a ROS system across multiple machines is not a hard task. Nevertheless, it is necessary to remember some basic ROS concepts; in particular that only one master can run on the system; therefore one has to decide which of them is the (only) master. Once that decision is taken the machines' environments should be configured correctly; it is necessary to update their `bash.rc` file to provide machines on which is not running the master a way to localize it. In the next lines an example about how that operation should be drafted is illustrated[9].

```
export ROS_MASTER_URI=http://{IP_ADDRESS_OF_REMOTE_PC}:11311

export ROS_HOSTNAME={IP_ADDRESS_CURRENT_PC}
```

In this part, a `roscore` and a `turtlesim_node` along with a teleoperation one will be launched on the master machine while on second will be execute another `turtlesim_node`.

```
roscore

rosrun turtlesim turtlesim_node __name:=master_turtle

rosrun turtlesim turtle_teleop_key
```

Listing 8.2: command to be launched on master machine

The remapping argument is used to clarify on which machine the nodes are running on.

```
rosrun turtlesim turtlesim_node __name:=node1_turtle
```

Listing 8.3: command to be launched on master machine

Once, the prior commands are executed on the two machines by running the `rqt_graph` command, one should be able to see a graph like the one in the following figure (8.7).

It is noteworthy that the two turtle's simulations are not independent, indeed even if the teleop node is executed just on one machine[10] it can control both the turtles. That happens because the two kinds of nodes publish and subscribe, respectively, on the `/turtle1/cmd_vel` topic. Messages published on that topic, regardless of which node publishes them, are delivered to every subscriber of that topic. That is the reason why the example could be extended to run on as many machines as want besides involving an indefinite number of `turtle_teleop_key` and `turtlesim_node` nodes. Getting back to details about publish-subscribe communication it is valuable to underline that none of the nodes explicitly

[9] At the following link: http://wiki.ros.org/ROS/Tutorials/MultipleMachines a complete description about multiple ROS machines initialization ca be found.

[10] It is not relevant on which machine the `turtle_teleop_key` node is running on.

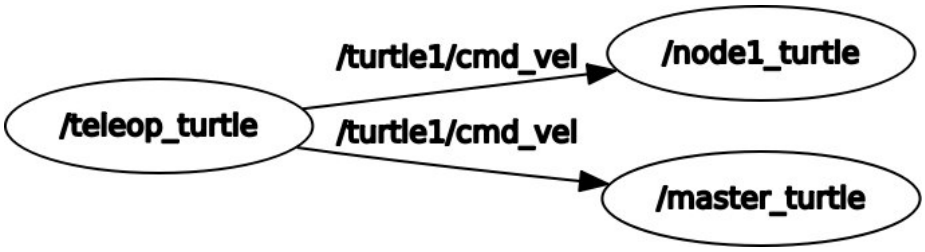


Figure 8.7: ROS rqt_graph showing multiple turtlesim nodes

know about any of the others; thus, their only interactions are indirect, at the level of topics and messages. Software as `turtle_teleop_key` that produces information can publish that information, without worrying about how that information is consumed, whereas software like `turtlesim_node` that consumes information can subscribe to the topic containing the data it needs, without worrying about how those data are produced. The independence of nodes along with the decomposition mechanism, is one of the key design features of ROS, as it facilitates larger tasks to be divided into smaller reusable pieces.

ROS mechanism for one-to-one communication are services, however, they are less common.

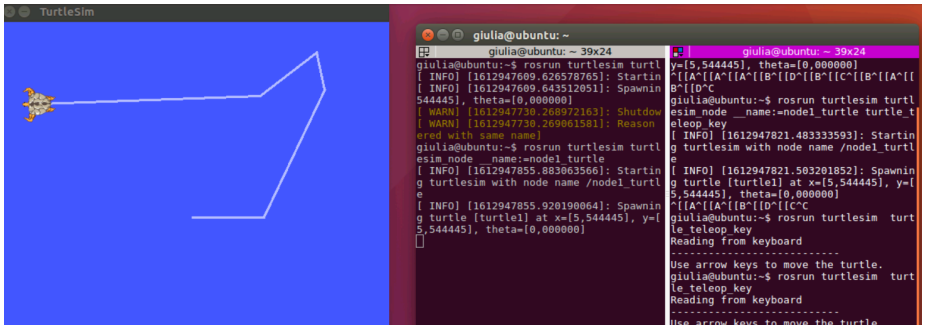


Figure 8.8: ROS turtlesim view running on different machines

# Chapter 9

# Navigation case use

## 9.1 Background

Moving around is one of the most common tasks a robot does to achieve this goal it must be able to localize itself and know which direction it should take. To gain this duty the device must integrate data from the map, localization system, sensors, and odometry to plan to calculate an efficient path plan from its actual location to its destination point.

Overall, the Navigation Stack working process can be described as follow: a navigation goal is sent to the Navigation Stack that computes the shortest traverse between the current position the destination one using the path-planning algorithm in the global planner. Next, the path is sent to the local planner that drives the robot along the path trying to avoid obstacles that are not present in the map using sensors' data. A new plan may be required from this latter planner to the global one if the robot got stuck until it can reach its target pose.

## 9.2 Introduction

In this section, the navigation task will be analyzed throughout two use cases: the first one is an introductory guide on Navigation Stack usage to accomplish common exploration task in the RViz environment; whereas the second shows a possible way to overcome the many-to-many communication provided in the previous chapter by publish-subscribe paradigm.

The following paragraph will be illustrated three ways to deal with the navigation task: providing the robot a "navigation goal" manually, using the teleoperation node, and letting it discover the world using the navigation algorithms.

## 9.3 Navigation goal

The first way to allow robots to move is by giving them a *navigation goal* manually using *RViz*, to do this the `move_base.xml` and `move_base_gmapping_5cm.launch` have to be executed as follow:

```
roslaunch navigation_stage move_base_gmapping_5cm.launch

roslaunch navigation_stage move_base.xml
```

Getting back to the details of the previously launched commands, the first one executes the `move_base_gmapping_5cm.launch` which contains the `move_base` node necessary to initialize the navigation's parameters, whereas the second is a launch file for running the navigation stack with gmapping at a map resolution of 5cm. It is valuable to notice that `roslaunch` command does not need the `roscore` command to be launched as it automatically detects if the system already has one running and if it is not so, it starts it.

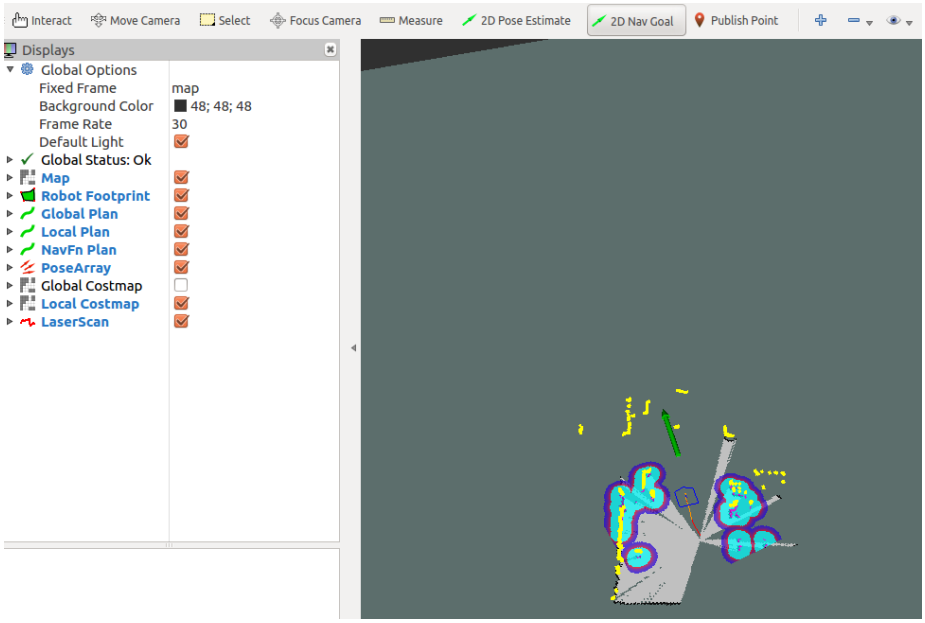


Figure 9.1: Rviz view of the robot navigation using navigation goal

Executing the commands in two different shell an RViz window should be opening, besides the world file of the map.

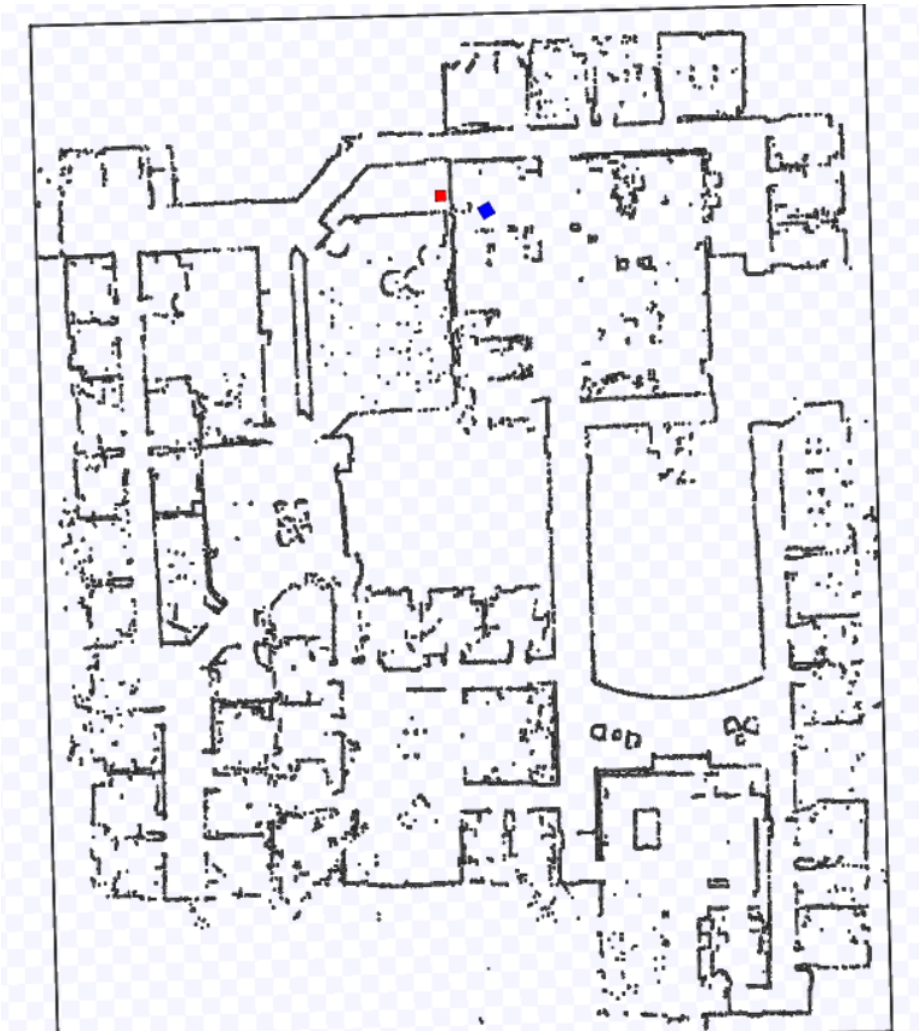

Figure 9.2: World plan used in the robot RViz simulation

The first one shows a robot shape inside the environment provided by the map highlighting in a different color the information related to the sensors along with other data.

The map, on the other hand, is not a simple image; it is an interactive world: it illustrates a plan of the static environment on one side, and on the other, it represents the robot dynamic position[1], which is automatically updated every time this latter moves. Furthermore, users can move the robot wherever they want inside the map dragging it with their mouses and the robot position in the

[1]The blue figure on the top of the map is a representation of the robot

RViz environment will be automatically updated.

Assuming the robot is properly localized, pressing on the *2D Nav Goal* button, then clicking and dragging in the RViz main stage, will be sent to the robot a destination position, known as *goal pose*, activating the computation process described in the background paragraph. The goal pose is represented by the red arrow in the lower right part of the map while the path to follow is outlined by an orange line. It is relevant to notice that ROS planner try to design a path which stays in low-cost area, namley as far as possible from walls (in yellow in the map) and any other obstacles (zoned in cold colors).

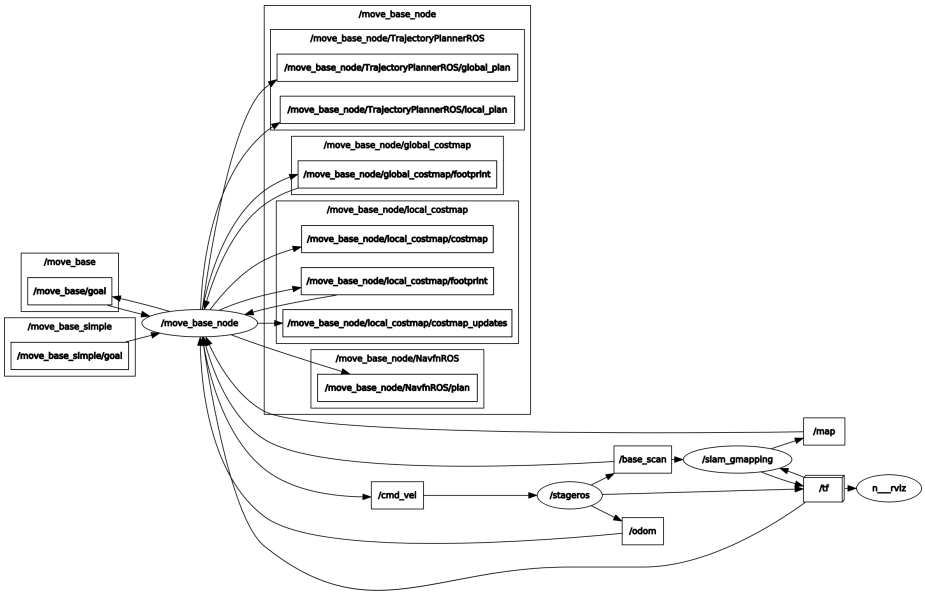


Figure 9.3: Rqt_graph of the robot navigation using navigation goals only

Furthermore, another remarkable feature that should be a highlight is that the robot does not *forget* the places it visited; indeed, it continues to update its map during the navigation.

The map can be saved at any moment for later uses by running the following command:

```
rosrun map_server map_saver -f my map
```

Notably, the previous command stores the map in a pair of files: the first encodes the occupancy data in an image (`.png` format) whereas the second is a `.yaml` file that describes the map metadata, and names the image file[2].

[2]For further information about map_server usage look at http://wiki.ros.org/map_server

## 9.4 Autonomous Navigation

Since it is not always convenient to provide navigation goals by hand, ROS offers several packages to perform autonomous navigation. This part will focus on two of them: the `frontier-navigation` and `explore-lite` packages underlining their key differences along with their similarities.

The leading purpose of both algorithms is to navigate an unknown environment without a human intervention building a map of it, nonetheless, the two methods use considerably different approaches that will be examined in detail in the following paragraphs.

### 9.4.1 Frontier-exploration

The frontier-exploration technique consists of finding frontiers, which are defined as the boundary between explored and unexplored territory. Notably, the user is asked to outline the area the robot is expected to explore, when this preliminary step is done the robot commences to navigate autonomously to frontiers to extend the *known zone* until the whole (reachable) space is explored. The computation of the path is mainly based on the cost of the frontiers; indeed they determine which of them will be visited next. As the robot pursues its traverse, new data about the surroundings comes in, leading to the formation of new frontiers; until there are no frontiers left, at that moment the exploration is concluded and the environment entirely mapped[85].

Getting back to details about frontier-exploration execution, one is asked to run the following commands:

```
roslaunch navigation_stage move_base_gmapping_5cm.launch

roslaunch navigation_stage move_base.xml

roslaunch frontier-exploration global_map.launch
```

Two of them were described in the previous sections while the last command is the one that performs the frontier-navigation.

**Rviz setup** Furthermore, since the algorithm needs the user to define the exploration area, this latter is asked to follow the next steps:

- opens a Marker plugin on RViz; by clicking on the *Add* button a popup window where the marker plugin is local should open.
- Pull down *Displays* →*Marker* →*Marker Topic* menu then select *exploration_polygon_marker* topic[3].
- to define the exploration region one is asked to design a polygon: one must press the *Publish Point* button at the top of RViz and next click on the map to define each of the polygon corners. Every time a new corner

[3]The *exploration_polygon_marker* will not be shown until the frontier exploration node is running, this is necessary to run all the commands before setup RViz environment.

is added one should see a blue line appear and connecting the last drawn corner with the previous one. It is important to notice that RViz requires the figure to be closed, consequently, the last corner has to be placed on the first one (as shown in the next figure.)

- once the figure is completed, to start the navigation one has to press the last time on the Publish Point button indicating a position within the polygon[55].

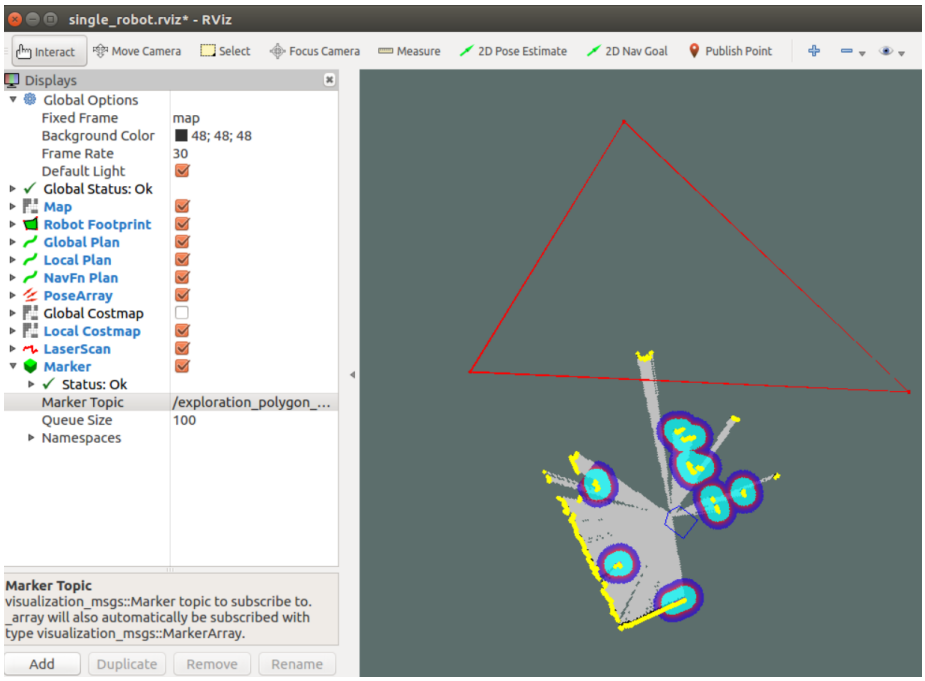


Figure 9.4: RViz view of the robot navigation using frontier exploration node

Analyzing `rqt_graph` one can notice what happen before and after having completed the figure.

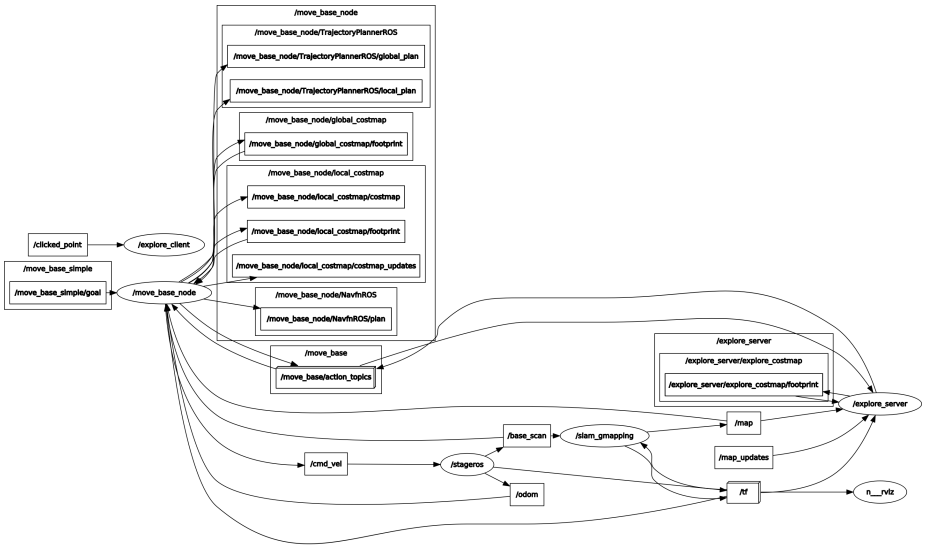


Figure 9.5: Rqt_graph of the robot navigation using frontier exploration algorithm without having started the autonomous navigation

In the first graph the topics `clicked_point` and `exploration_polygon_marker` are not present along with `explore_client` while in the second can be found at the top of the graph.

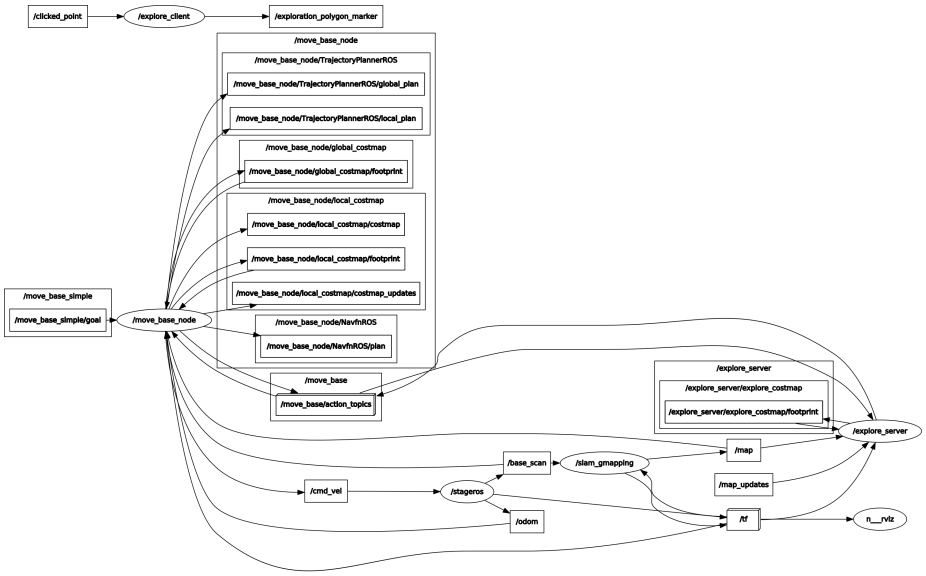


Figure 9.6: Rqt_graph of the robot navigation using frontier exploration algorithm activate the autonomous navigation

### 9.4.2 Explore-lite

The second method examined is based on a greedy frontier-based exploration, that requires no user intervention. Notably, the robot's greedily traverse will start autonomously and continue until no new frontiers could be found.

This package is particularly efficient in terms of resources used because unlikely the previous one it does not create its own costmap; the node simply subscribes to `nav_msgsOccupancyGrid` messages while movement commands will be send to `move_base`.

```
roslaunch navigation_stage move_base_gmapping_5cm.launch

roslaunch navigation_stage move_base.xml

roslaunch explore_lite global_map.launch
```

As in the previous happened in the previous example, the first two commands are the same, since they are required to setup the Navigation environment, while the last one it is the command which activate explore-lite node[54].

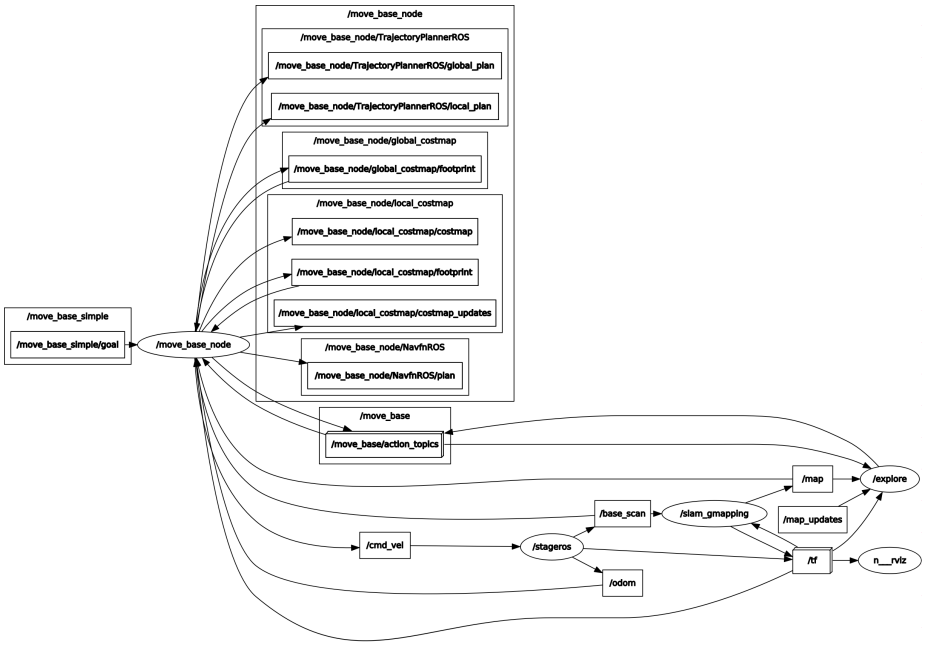


Figure 9.7: Rqt_graph of the robot navigation using explore_lite node

### 9.4.3 Teleoperation

This latter part uses files contained in the project repository, which is assumed as installed on users pc, besides the project path `devel.bash` is expected to be sourced in `.bashrc`. Indeed, if the path is not sourced among the environment variables one should remember to launch the following command (from the devel folder[4]) in each terminal before running the next example:

```
source devel/setup.bash
```

The last method to control robots that will be studied is teleoperation. That last technique has already been discussed in the former chapter, however, in this one way to overcome publish-subscribe multi-to-multi communication will be shown. Furthermore, this latter use case involves real robots simulation: Turtlebot will be run in a Gazebo environment.

Firstly, will be provided the three commands to run, next will analyze each of them.

[4]The devel folder, as well as the build folder, are not present in the repository as one download it, they only appear when catkin_make command is launched from inside ros_ws folder. One would not be able to execute the repository code until the build system has not created the directories.

```
roslaunch turtlebot3_ds my_multi_turtlebot3.launch

roslaunch turtlebot3_ds turtlebot3_teleop_key_tb3_0.launch

roslaunch turtlebot3_ds turtlebot3_teleop_key_tb3_1.launch
```

The first command runs the Gazebo node and it initializes the parameters related to the world and robots' position within it. Notably, the file launches two nodes: the `spawn_urdf` node load the Turtlebot model[5] inside the world, while the `robot_state_publisher` broadcast robot states in ROS environment. As a result of that first operation, one should see a Gazebo window opening showing two Turtlebot robots within a house.

```
<?xml version="1.0" ?>
<launch>
  <!--Arguments-->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model
    type [burger, waffle, waffle_pi]"/>
  <arg name="move_forward_only" default="false"/>

  <!--Turtlebot' names-->
  <arg name="first_tb3"  default="tb3_0"/>
  <arg name="second_tb3" default="tb3_1"/>

  <!--First Turtlebot position -->
    <!--Cartesian coordinates-->
  <arg name="first_tb3_x_pos" default="-1.2"/>
  <arg name="first_tb3_y_pos" default="0.0"/>
  <arg name="first_tb3_z_pos" default=" 0.0"/>
    <!--rotation along Z axis-->
  <arg name="first_tb3_yaw"   default=" 0.0"/>

  <!--Second Turtlebot position -->
    <!--Cartesian coordinates-->
  <arg name="second_tb3_x_pos" default=" 1.2"/>
  <arg name="second_tb3_y_pos" default="0.0"/>
  <arg name="second_tb3_z_pos" default=" 0.0"/>
    <!--rotation along Z axis-->
  <arg name="second_tb3_yaw"   default=" 0.0"/>

  <param name="/use_sim_time" value="true"/>

 <!--find and load turtlebot3 model-->
  <param name="robot_description" command="$(find xacro)/xacro --
    inorder $(find turtlebot3_ds)/turtlebot3_description_ds/urdf/
    turtlebot3_$(arg model).urdf.xacro" />

   <!-- Spawn Turtlebot0 -->
  <group ns = "$(arg first_tb3)">
    <!-- Node responsible to broadcast the state of a robot-->
    <node pkg="robot_state_publisher" type="robot_state_publisher"
    name="robot_state_publisher" output="screen">
      <param name="publish_frequency" type="double" value="50.0" />
```

[5]In ROS elements of robots are described using the Unified Robotic Description Format (URDF); which is an XML file format.

```
      <param name="tf_prefix" value="$(arg first_tb3)" />
       <!--Specify at which Turtlebot the node refers to-->
    </node>
    <!--locate the robot in right position-->
    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
    args="-urdf -model $(arg first_tb3) -x $(arg first_tb3_x_pos) -
    y $(arg first_tb3_y_pos) -z $(arg first_tb3_z_pos) -Y $(arg
    first_tb3_yaw) -param /robot_description" />
</group>

  <!-- Spawn Turtlebot1 -->
     <group ns = "$(arg second_tb3)">
        <!-- Node responsible to broadcast the state of a robot-->
        <node pkg="robot_state_publisher" type="
    robot_state_publisher" name="robot_state_publisher" output="
    screen">
          <param name="publish_frequency" type="double" value="50.0
    " />
          <param name="tf_prefix" value="$(arg second_tb3)" />
       <!--Specify at which Turtlebot the node refers to-->
        </node>
      <!--locate the robot in right position-->
      <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
    args="-urdf -model $(arg second_tb3) -x $(arg second_tb3_x_pos)
     -y $(arg second_tb3_y_pos) -z $(arg second_tb3_z_pos) -Y $(arg
     second_tb3_yaw) -param /robot_description" />
  </group>


  <!-- Start gazebo and load the world -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <!-- Load the model of the right world -->
      <arg name="world_name" value="$(find turtlebot3_ds)/worlds/
    turtlebot3_house.world"/>
      <arg name="paused" value="false"/>
      <arg name="use_sim_time" value="true"/>
      <arg name="gui" value="true"/>
      <arg name="headless" value="false"/>
      <arg name="debug" value="false"/>
    </include>
</launch>
```

Listing 9.1: turtlebot3_ds my_multi_turtlebot3.launch code file

The second and the third commands aim to run the `teleop_twist_keyboard` node to control the robots. One relevant aspect of the file is that allows users to specify which robot the node is in-charge of. Overcoming the limitation of the multi-to-multi communication of the Publish-Subscribe pattern.

```
<?xml version="1.0" ?>
<launch>
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model
    type [burger, waffle, waffle_pi]"/>
  <param name="model" value="$(arg model)"/>

  <!-- turtlebot3_teleop_key already has its own built in velocity
    smoother -->
```

```
  <node pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py"
     name="teleop_twist_keyboard_tb3_0" output="screen" arg="tb3_0"
    >
<remap from="cmd_vel" to="tb3_0/cmd_vel"/>
</node>
</launch>
```

Another significant aspect that should take into account is that the launch file method provide a light and simple way to run a several nodes with just on command. Furthermore, it let the user to make the example as complex as this wants by adding as much robots as one wants and controlling them with autonomous driving algorithms or teleoperation nodes.

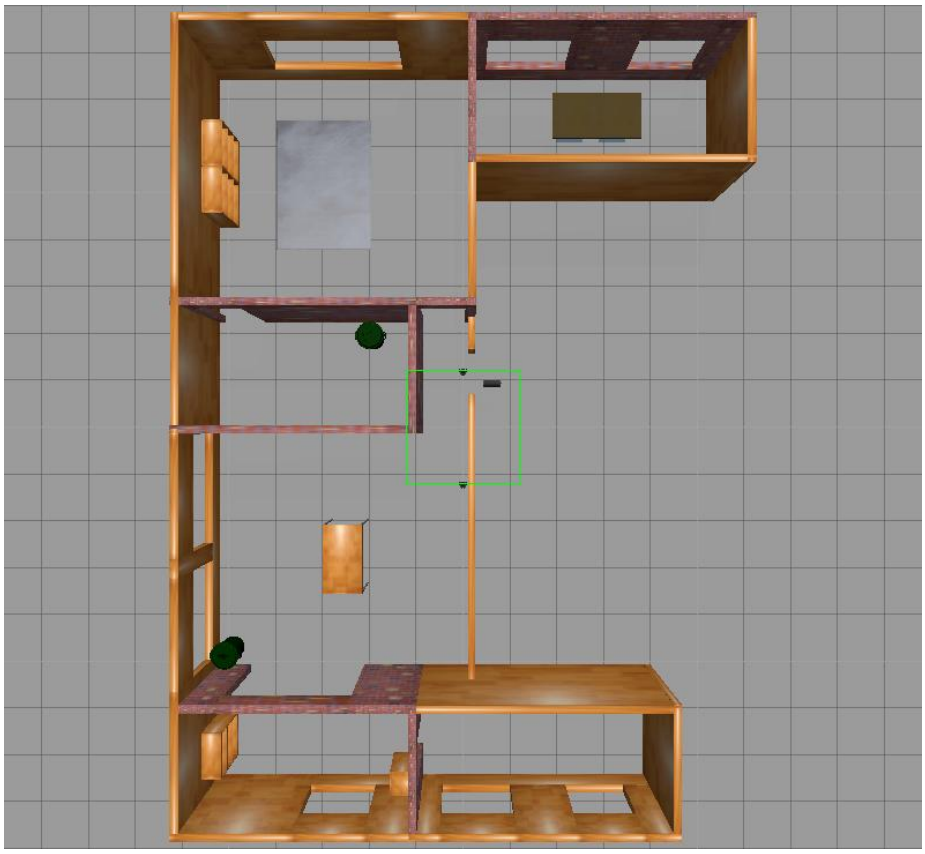

Figure 9.8: Gazebo simulation showing multiple Turtlebot3 in Turtlebot's house

To conclude, it is relevant to underline that after the a preliminary step of environment setup, the three `.launch` files can be run on different machines as shown in paragraph 8.2.3 and it can be extended by adding several other nodes, as the ones related to autonomous navigation, to the configuration files.

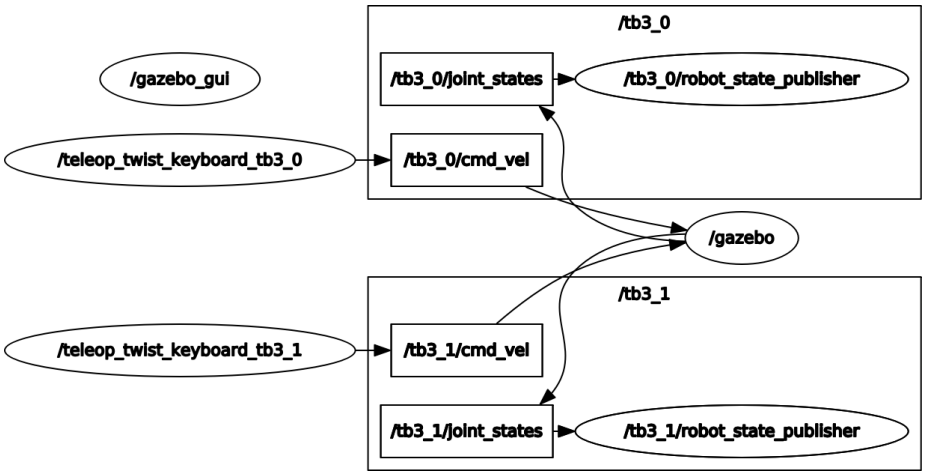


Figure 9.9: rqt_graph of the multiple turtlebots example

## 9.5 Beyond the project

This last use case chapter provides an overview of ROS navigation stack usage besides introducing the principal tools used for simulation in the ROS environment, namely Gazebo and RViz. In particular, it focuses on showing different kinds of navigation using several tools and devices.

Nevertheless, it is valuable to underline that all the tasks and navigation methods can be implemented on the turtlebot platform. In this respect, it is significant to notice that the ROS manual suggests ROBOTIS10.2 repository as the target repository to use turtlebot3 robots: https://github.com/ROBOTIS-GIT/turtlebot3. By installing, the aforementioned repository one could run a single turtlebot analyzing its behaviour in Gazebo by executing the following commads.

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

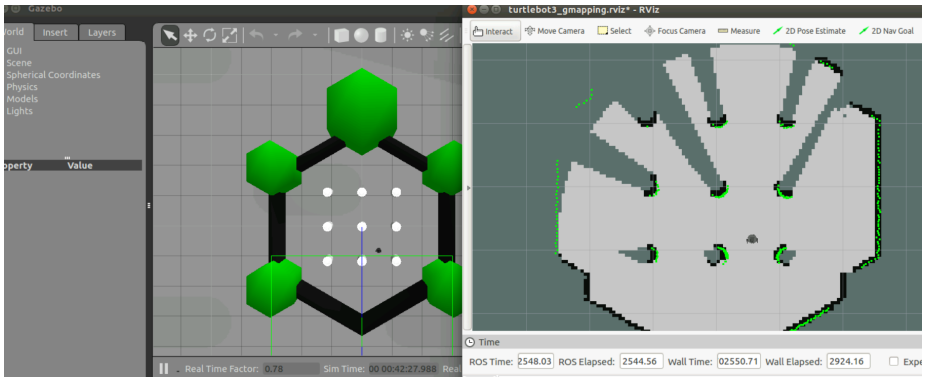


Figure 9.10: View of Gazebo and RViz performing single turtlebot navigation

Furthermore, to perform autonomous navigation and analyze robotphysical structure on RViz, the following files should be executed.

```
roslaunch turtlebot3_navigation move_base.launch

roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=
    gmapping
```

In the light of the foregoing it should be clear, the first command executes the `move_base` node, while the second the slam algorithm used for the navigation; especially in this former case, the file requires to specify which kind of method should be used for the navigation, this is the reason why the gmapping parameter is passed as the second argument to the roslaunch command.

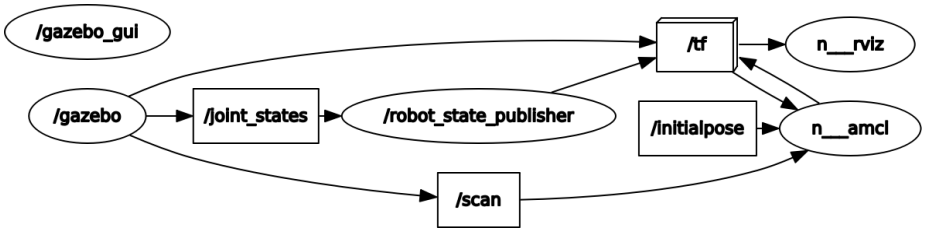


Figure 9.11: Rqt_graph showing the initial nodes of the example

At this stage the RViz window should be opened and allow users to use navigation nodes to perform the task, however, if one would prefer to perform autonomous navigation then one should launch:

```
roslaunch turtlebot3_slam turtlebot3_frontier_exploration.launch
```

to perform the exploration using the frontier algorithm, or executing the following code to control the robot motion with the teleoperation node.

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

In the next lines the evolution of system graph in both the previous exposed cases will be illustrated.

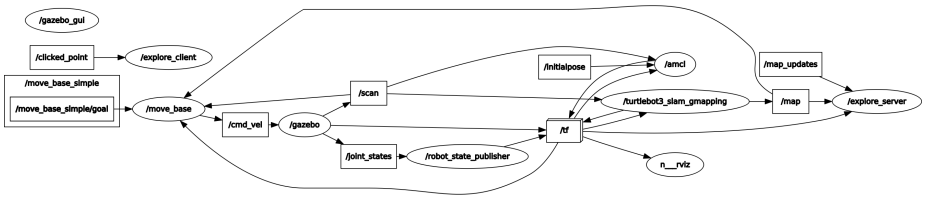


Figure 9.12: Rqt_graph showing the evolution of system once the frontier node is added

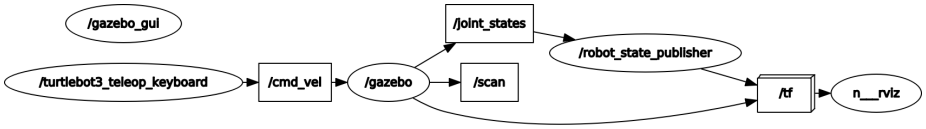


Figure 9.13: Rqt_graph showing the evolution of system once the teleoperation node is added

## 9.6 Conclusion

The illustrated study case is only the thin end of the wedge, there would be more applications of the Navigation Stack and possible ways to increase the complexity of the examples nevertheless since this project aims not to do so this section terminates with some brief usage overview.

# Chapter 10

# Conclusions

## 10.1 Recap

The project aimed to study ROS middleware along with ROS Navigation Stack on one hand, and to provide a report about their main functionalities and usage case on the other. Although ROS is a performing platform for robotics applications, there is no doubt that its intricacy makes it hard to learn it from the default wiki pages. Indeed, one of the most important challenges was finding sources on which to base documentation. Furthermore, another crucial aspect that made it even more challenging to write the guide was starting with ROS simulation programs worked, as Gazebo, the major simulator in ROS has no inbuilt features to program, coding in ROS is the only way to defines the simulation working process.

The project tackles the issue of providing a report of ROS middleware focused on its general characteristics and architecture besides a description of one of its main frameworks. Notwithstanding, the report is not a complete guide for ROS middleware because of the platform complexity and extension; this latter is only a concise guide for ROS beginners, which provides besides the platform analysis also a set of meaningful examples of ROS usage with a detailed explanation of their working processes. Furthermore, it introduces some relevant tool to perform realistic simulations in ROS environment, Gazebo and RViz, besides providing programs, as RQt, to understand ROS nodes communication.

## 10.2 Future Works

ROS is a widespread platform, whose number of applications increase day by day, therefore thinking of a future of work related to it, it is not a trivial task, since it could be about almost anything ROS-related: a new ROS stack study, a closer analysis of the Navigation one, or also a new case study on ROS 2 for instance. Among the most interesting future works one could develop there are: ROS testing on a real hardware platform, besides the study and the relative

experimentation of ROS2. Indeed, despite ROS tools for simulations performs quite well, I believe that developing a real *cyber-physical* project would far more engaging. Nevertheless, that project would involve a more comprehensive understanding of the ROS modeling process, especially for what concerns robot modeling with `URDF` files and ROS `tf` package, which is not trite at all.

# LOGO legacy

*Logo is the name for a philosophy of education and a continually evolving family of programming languages that aid in its realization.*
Harold Abelson Apple Logo, 1982

## LOGO

LOGO is an educational computer programming language created by Papert in 1967, the main purpose of the project was to develop a tool that enables young children to code. LOGO is a dialect of Lisp whose main features are: modularity, extensibility, interactivity, and flexibility; it is designed to have a "low threshold and no ceiling" since it is accessible to novices, including young children, and also supports complex explorations and sophisticated projects by experienced users as one can use it for a wide range of activities such as mathematics, language, music, robotics, telecommunications, and science. The most popular Logo environments have involved the Turtle, originally a robotic creature that sat on the floor and could be directed to move around by typing commands at the computer. Soon the Turtle migrated to the computer graphics screen where it is used to draw shapes, designs, and pictures.

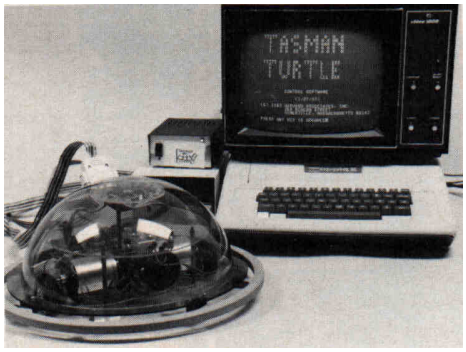


Figure 10.1: Logo

Some turtle species can change shape to be birds, cars, planes, or whatever

the designer chooses to make them. In Logo environments with many such turtles, or "sprites" as they are sometimes called, elaborate animations and games are created. Among his works two stand out for their relevance: the development of the LOGO programming language and the Constructionist theory.

## Seymour Papert

Seymour Papert is a South African-born American mathematician, computer scientist, and educator, who co-founded the MIT Artificial Intelligence Laboratory with Marvin Minsky, wherein in 1967 created the first version of Logo 1967. Papert worked on a vast set of projects in many different fields; however, two of them stand out for their relevance: the first one is the LOGO creation (which importance has already been discussed in the previous paragraph) and the second is its Constructionism, both of them are well described in his major book Mindstorms. Besides these, another quite popular project is the development of the LEGO Mindstorms kit.

## A Logo Primer

The most popular Logo environment has involved the Turtle, originally a robotic creature that moved around on the floor.

Student at computer It can be directed by typing commands at the computer. The command 100 causes the turtle to move forward in a straight line of 100 "turtle steps". Right 45 rotates the turtle 45 degrees clockwise while leaving it in the same place on the floor. Then forward 50 causes it to go forward 50 steps in the new direction.

With just the two commands forward and right, the turtle can be moved in any path across the floor. The turtle also has a pen that may be lowered to the floor so that a trace is left of where it has traveled. With the pen down, the turtle can draw geometric shapes, and pictures, and designs of all sorts.

# ROS

## Actionlib

In this section will be given some basic information about the actionlib stack, which provides a standardized interface for interfacing with preemptable tasks, for further information looks at http://wiki.ros.org/actionlib/DetailedDescription.

### Overview

In large systems, someone may want to send a request to a node to perform some task, and also receive a reply to the request; ROS services can handle these kinds of tasks. Nevertheless, if the task takes a long time to be executed by the service, it would be useful to be able to cancel the request during execution or to receive periodic updates about the request status. The actionlib package allows users to create servers that execute long-running tasks (that can be preempted) besides a client interface to be able to send requests to the server.

#### Client-Server Interaction

The Action Client and Action Server use a "ROS Action Protocol" to communicate, which is built on top of ROS messages, in addiction they provide a simple API for call and callbacks functions to allow users to request goals (on the client side) or to execute goals (on the server side).

#### Action Specification

To enable the communication between the client and server, few messages on which they communicate must be defined with an action specification. In the following paragraph will be presented the message defined by the action specification which is: the Goal, Feedback, and Result messages.

**Goal** : the notion of the goal must be defined to accomplish task actions; this can be sent from the Action Client to send new goal messages to the Action Server.

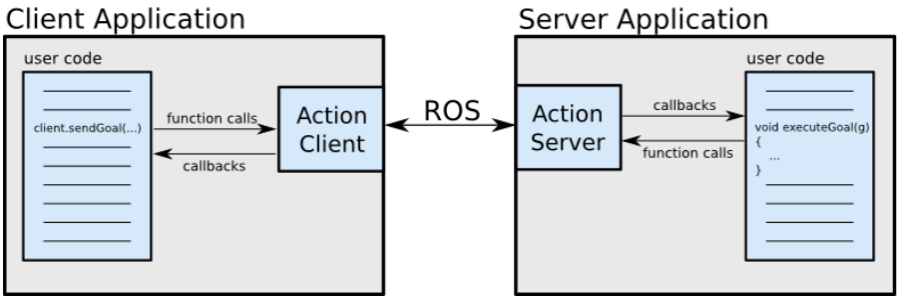


Figure 10.2: ROS schema of the actionlib Client-Server interaction

**Cancel** : this message can be sent by the Action Client to the Action Server to cancel a request.

**Status** : Action Server can send this kind of message to the Action Client to notify them of the current state of every goal in the system.

**Feedback** : it is a type of message used by the server implementer to give to the Action Client auxiliary information for a goal.

**Result** : it is how the Action Server can inform the Action Client that it has completed the goal. The main difference between this kind of message and the one presented in the previous paragraph is that result messages are sent exactly once[50] [51].

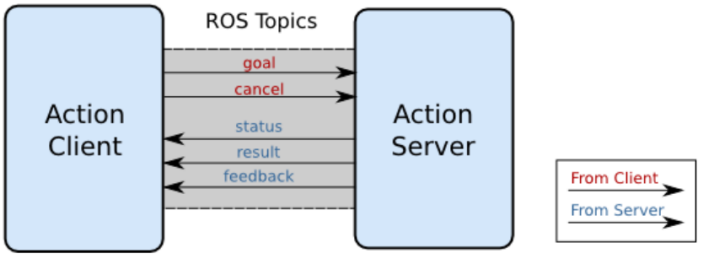


Figure 10.3: ROS actionlib: Action interface schema

# Catkin

## Overview

`catkin` is the official build system of ROS starting from *ROS Groovy* and the successor to the original ROS build system, `rosbuild`. catkin combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow (improved automatic dependencies management and compilation of large project)[12].

catkin was designed to be more conventional than `rosbuild`, allowing for better distribution of packages, better cross-compiling support, and better portability. catkin's workflow is very similar to `CMake`'s but adds support for automatic 'find package' infrastructure and building multiple, dependent projects at the same time.[47]

The name catkin comes from the tail-shaped flower cluster found on willow trees[1].

the catkin build system is organized in a workspace containing different spaces and packages, this feature is very useful for having a common files/directory structure and for building multiple packages with complex dependencies.


Figure 10.4: ROS Catkin: main folder description

For further information about the catkin build system look at the following URL http://wiki.ros.org/catkin/conceptual_overview.

## Why Does ROS Have a Custom Build System?

ROS is a very large collection of loosely federated packages. That means lots of independent packages which depend on each other, utilize various programming languages, tools, and code organization conventions. Because of this, the build process for a target in some packages may be completely different from the

[1] a reference to Willow Garage where catkin was created

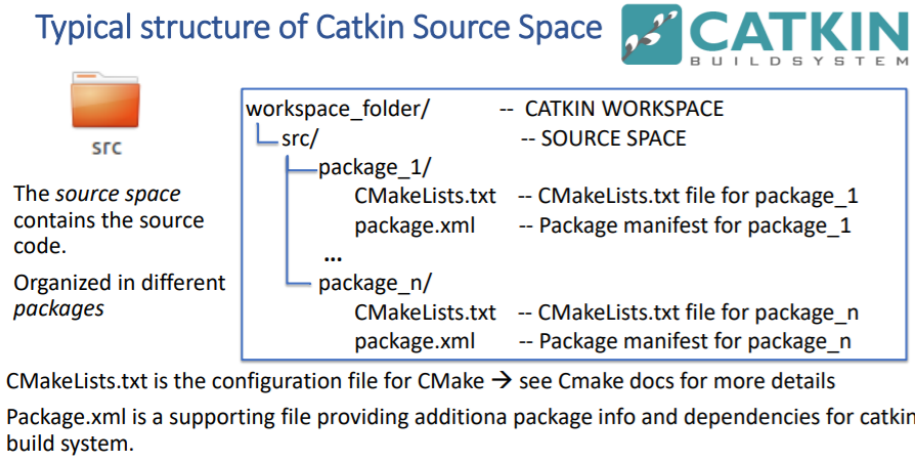


Figure 10.5: ROS Catkin: typical structure of catkin Source Space

way another target is built. catkin specifically tries to improve development on large sets of related packages consistently and conventionally. In other words, both rosbuild and now catkin aim to make building and running ROS code easier by using tools and conventions to simplify the process. Efficiently sharing ROS-based code would be more difficult without it.

### Decoupling from ROS

One of the philosophies behind ROS is to minimize the number of ROS-specific tools needed to create, manage, and utilize ROS packages and to always try to defer to well-established, widely-used, third-party, open-source tools (e.g. using libtinyxml instead of writing a custom XML parser) instead. Catkin is independent of the ROS ecosystem and can even be used on non-ROS projects. That means catkin lets you easily mix your codebase with non-catkin projects. Catkin adds a lot of features on top of vanilla CMake that make it an appealing development tool even for non-ROS-related projects.

## Publish Subscribe

### Introduction

The publish/subscribe pattern is a well know interaction paradigm that allows subscribers to express their interest in one or more events and be notified if any event, generated by a publisher, matches their registered interest.

The basic system model for publishing/subscribe interaction relies on an event notification service providing storage and management for subscriptions

and efficient delivery of events.

Subscribers register their interest in events by calling a subscribe() operation on the event service, without knowing the effective sources of these, moreover, they can use symmetric operation unsubscribe() to terminate a subscription. It is important to underline that the subscription information remains stored just in the event service and is not forwarded to publishers.

Symmetrically, publishers can generate events by calling a publish() operation, subsequently, the event service propagates the event to all relevant subscribers (every subscriber will receive an event for every event conforming to its interest).

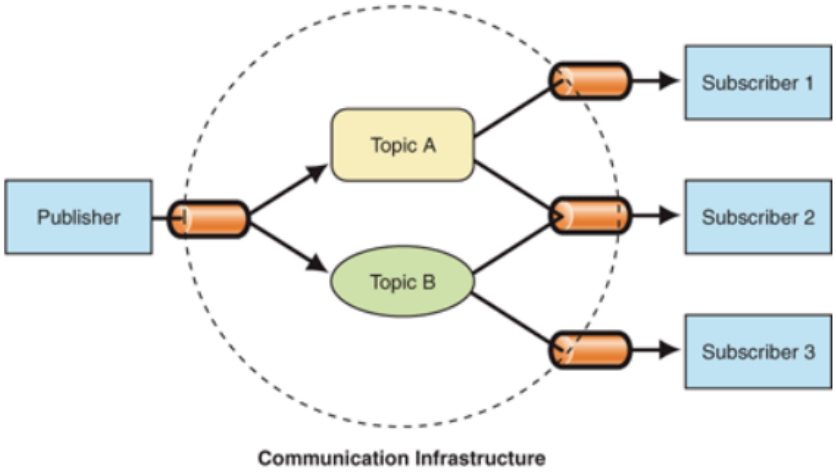


Figure 10.6: Publish subscriber pattern interaction's graph

## Decoupling

The decoupling that the event service provides between publishers and subscribers can be decomposed along the following three dimensions.

- **Space decoupling**: the interacting parties do not need to know each other; the publishers do not hold references to the subscribers or know how many of them are, the speculate situation happens to the subscribers.
- **Time decoupling**: the interacting parties do not need to be actively participating in the interaction at the same time. In particular, the publisher might publish some events while the subscriber is disconnected, and conversely, the subscriber might get notified about the occurrence of some event while the original publisher of the event is disconnected.

- **Synchronization decoupling**: publishers are not blocked while producing events, and subscribers can get asynchronously notified of the occurrence of an event while performing some concurrent activity. Decoupling these operations increases scalability by removing all explicit dependencies between the interacting participants[17].

# Turtlebot

## Willow Garage

*Willow Garage* was a robotics research lab and technology incubator started in late 2006 by Scott Hassan aiming to develop hardware and open source software for personal robotics applications[90].

In January 2007, the company hired its first employees: Jonathan Stark, Melonee Wise, Curt Meyers, and John Hsu to work at the first projects of the Willow Garage, among which there was also an SUV entrant into the DARPA Grand Challenge and an autonomous solar-powered boat for deploying scientific payloads in open oceans. The next year, Eric Berger and Keenan Wyrobek pitched Willow Garage on collaborating a common hardware (PR1) and software (ROS) platforms and the idea of creating a Personal Robotics Program at Willow Garage.

Willow Garage continues to work on a large number of projects until it shut down in early 2014[91].


Figure 10.7: Willow Garage logo

## ROBOTIS

*ROBOTIS*, is an enterprise founded in 1999 whose core business are robots. Indeed, since its birth ROBOTIS company has contributed to commercialize personal robots that can work in a variety of specialized fields ranging from spreading and sharing technology among robot developers. In particular, the company focus on developing robotic solutions that are modularized

on one hand and standardized on the other. Furthermore, the company runs ROBOTISKIDSLAB institute that specializes in robots and coding education and it also provides solutions for professionals which include the actuator, controller, and robot platform[39].


Figure 10.8: Robotis logo

# Bibliography


[1] Evan Ackerman. Turtlebot inventors tell us everything about the robot, 2013. Available at https://spectrum.ieee.org/automaton/robotics/diy/interview-turtlebot-inventors-tell-us-everything-about-the-robot.

[2] Robin Amsters and Peter Slaets. Turtlebot 3 as a robotics education platform, 01 2020.

[3] Madhur Behl. Ros filesystem and workspaces. F1/10 autonomous racing, University of Virginia, 2019.

[4] Madhur Behl. Understanding ros using turtlesim. F1/10 autonomous racing, University of Virginia, 2019.

[5] Gloria Beraldo. Sviluppo di un software per il controllo di un robot di telepresenza via brain-computer interface. Master's thesis, Università degli Studi di Padova, 2014.

[6] Brian Bergstein. Inventors (2013). Available at https://www.technologyreview.com/innovator/morgan-quigley/.

[7] Adrian Canoso. Ros gui, 2012. Available at http://web.archive.org/web/20130518142837/http://www.willowgarage.com/blog/2012/10/21/ros-gui.

[8] The Construct. Why i choose ros for my master thesis, 2019. Available at https://www.theconstructsim.com/why-i-choose-ros-for-my-master-thesis/.

[9] The Constructsim. Ros 2 vs. ros 1 : Which one is better for me? Available at https://www.theconstructsim.com/infographic-ros-1-vs-ros-2-one-better-2/.

[10] Valter Costa, Rosaldo Rossetti, and Armando Sousa. Simulator for teaching robotics, ros and autonomous driving in a competitive mindset, 10 2017.

[11] Lorenzo Croccolino. La piattaforma ros per lo sviluppo di applicazioni robotiche: panoramaica e caso di studi. Master's thesis, Alma Mater Studiorum Università di Bologna, 2018-2019.

[12] Diego Dall'Alba. Introduction to ros. Altair robotics lab nta3, Università di Verona, 2020.

[13] ROS 2 Design. *Why ROS 2?* Available at https://design.ros2.org/articles/why_ros2.html.

[14] Ilia Dobriakov. Design and development of a ros based lidar platform to scan environment for a mobile robot's autonomous motion. Master's thesis, Saimaa University of Applied Sciences, 2019.

[15] ROBOTS e Manual. *Turtlebot3-Overview.* Available at https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

[16] Pablo Estefó, Jocelyn Simmonds, Romain Robbes, and Johan Fabry. The robot operating system: Package reuse and community dynamics, 02 2019.

[17] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, 06 2003.

[18] LOGO FOUNDATION. A logo primer. Available at https://el.media.mit.edu/logo-foundation/what_is_logo/logo_primer.html.

[19] LOGO FOUNDATION. What is logo. Available at https://el.media.mit.edu/logo-foundation/what_is_logo/index.html.

[20] Lorenzo Galtarossa. Obstacle avoidance algorithms for autonomous navigation system in unstructured indoor areas. Master's thesis, Politecnico di Torino, 2018.

[21] gazebosim.org. *Gazebo.* Available at http://gazebosim.org/.

[22] gazebosim.org. *Gazebo Architecture.* Available at http://gazebosim.org/tutorials?tut=architecture&cat=get_started.

[23] Khan Saad Bin Hasan. What, why and how of ros, 2019. Available at https://towardsdatascience.com/what-why-and-how-of-ros-b2f5ea8be0f3.

[24] Luca Iocchi. Actions and plans. Technical report, Sapienza Università di Roma, 2014.

[25] Serena Ivaldi, Vincent Padois, and Francesco Nori. Tools for dynamics simulation of robots: a survey based on user feedback, 02 2014.

[26] Lentin Joseph. *Mastering ROS for Robotics Programming.* PACKT publishing, 2015.

[27] Fiki Jusuf. Auto-navigation for robots. implementation of ros. Master's thesis, Helsinki Metropolia University of Applied Sciences, 2016.

[28] Anis Koubaa. *Robot Operating System (ROS), The Complete Reference (Volume 1)*. Janusz Kacprzyk, Polish Academy of Sciences, 2019.

[29] Davide Leardini. La piattaforma ros per lo sviluppo di applicazioni robotiche. Master's thesis, Alma Mater Studiorum Università di Bologna, 2013-2014.

[30] Rodrigo Longhi, Guimaraes, Andre Schneider de Oliveira, Joao Fabro, Thiago Becker, and Vinıcius Amilgar Brenner. Ros navigation: Concepts and tutorial. Technical report, Federal University of Technology, 2016.

[31] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2, 10 2016.

[32] Michele. Thanks seymour papert for changing the world, 08 2016.

[33] Mayank Mittal. Introduction to robot simulation (gazebo). Ae640a: Autonomous navigation, Indian Institute of Technology Kanpur, 2018.

[34] Jason M. O'Kane. *A Gentle Introduction to ROS*. Independently published, October 2013. Available at http://www.cse.sc.edu/~jokane/agitr/.

[35] Gianluca Palli. Introduction to ros programming. Industrial robotics m course, Alma Mater Studiorum Università di Bologna, 2020-2021.

[36] Devendra Patil, Ming Xie, and Sai Velamala. Development of ros-based gui for control of an autonomous surface vehicle, 12 2017.

[37] Cesare Pecone. Ros over the internet: developing a vpn-based ros network to enable remote communication using a 3d printed robot hand. Master's thesis, Politecnico di Torino, 2018.

[38] Stelios Piperakis. Introduction to ros. Autonomous robotic navigation, University of Crete, 2019.

[39] pitchbook. Robotis general information. Available at https://pitchbook.com/profiles/company/111523-42#overview.

[40] YoonSeok Pyo, HanCheol Cho, RyuWoon Jung, and TaeHoon Lim. *ROS Robot Programming)*. ROBOTIS Co.,Ltd., 2017.

[41] Morgan Quigley, Eric Berger, and Andrew Y. Ng. Stair: Hardware and software architecture. Technical report, Stanford University, 2007.

[42] Morgan Quigley, Eric Berger, Andrew Y. Ng., Brian Gerkey, Josh Faust, Ken Conley, Tully Foote, Jeremy Leibs, and Rob Wheele. Ros: an open-source robot operating system. Technical report, Stanford University and Willow Garage and University of Southern California, 2009.

[43] Morgan Quigley, Brian Gerkey, and William D. Smart. *Programming Robots with ROS*. O'Reilly Media, In, 2015.

[44] Marco Reschiglian. Sviluppo e implementazione di una interfaccia grafica in realtà aumentata per il controllo semplificato di sistemi robotici da remoto o in ambiente ostile. Master's thesis, Università degli Studi di Padova, 2017.

[45] Open Robotics. Overview and usage of rqt. Available at https://docs.ros.org/en/foxy/Tutorials/RQt-Overview-Usage.html.

[46] Generation Robots. Ros – robot operating system, 2016.

[47] ROS.org. *catkin conceptual overview*. Available at http://wiki.ros.org/catkin/conceptual_overview.org/.

[48] ROS.org. *geometry_msgs/Twist Message*. Available at http://docs.ros.org/en/jade/api/geometry_msgs/html/msg/Twist.html.

[49] ROS.org. *Naming Conventions for Catkin Based Workspaces*. Available at https://www.ros.org/reps/rep-0128.html.

[50] ROS.org. *ROS actionlib*. Available at http://wiki.ros.org/actionlib.

[51] ROS.org. *ROS actionlib Detailed Description*. Available at http://wiki.ros.org/actionlib/DetailedDescription.

[52] ROS.org. *ROS-Concepts*. Available at http://wiki.ros.org/ROS/Concepts.

[53] ROS.org. *ROS Environment Variables*. Available at http://wiki.ros.org/ROS/EnvironmentVariables#ROS_IP.2BAC8-ROS_HOSTNAME.

[54] ROS.org. *ROS explore_lite*. Available at http://wiki.ros.org/explore_lite.

[55] ROS.org. *ROS frontier_exploration*. Available at http://wiki.ros.org/frontier_exploration.

[56] ROS.org. *ROS-Introduction*. Available at http://wiki.ros.org/ROS/Introduction.

[57] ROS.org. *ROS Kinetic Kame*. Available at http://wiki.ros.org/kinetic.

[58] ROS.org. *ROS-Messages*. Available at http://wiki.ros.org/Messages.

[59] ROS.org. *ROS Navigation*. Available at http://wiki.ros.org/navigation.

[60] ROS.org. *ROS Network Setup*. Available at https://wiki.ros.org/ROS/NetworkSetup#Single_machine_configuration.

[61] ROS.org. *ROS roscore*. Available at http://wiki.ros.org/roscore.

[62] ROS.org. *ROS roslaunch*. Available at http://wiki.ros.org/roslaunch.

[63] ROS.org. *ROS roslaunch XML*. Available at http://roswiki.autolabor.com.cn/roslaunch(2f)XML.html.

[64] ROS.org. *ROS-TCPROS*. Available at http://wiki.ros.org/ROS/TCPROS.

[65] ROS.org. *ROS TF*. Available at http://wiki.ros.org/tf.

[66] ROS.org. *ROS turtlesim*. Available at http://wiki.ros.org/turtlesim.

[67] ROS.org. *ROS-Tutorial-Building a ROS Package*. Available at http://wiki.ros.org/ROS/Tutorials/BuildingPackages.

[68] ROS.org. *ROS-Tutorial-Creating a ROS Package*. Available at http://wiki.ros.org/ROS/Tutorials/CreatingPackage.

[69] ROS.org. *ROS-Tutorial-Creating a workspace for catkin*. Available at http://wiki.ros.org/catkin/Tutorials/create_a_workspace.

[70] ROS.org. *ROS-Tutorial-Creating Msg And Srv*. Available at http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv.

[71] ROS.org. *ROS-Tutorial-Examining Publisher Subscriber*. Available at http://wiki.ros.org/ROS/Tutorials/ExaminingPublisherSubscriber.

[72] ROS.org. *ROS-Tutorial-Writing Publisher Subscriber(python)*. Available at http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29.

[73] ROS.org. *ROS Tutorials Installing and Configuring Your ROS Environment*. Available at http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment.

[74] ROS.org. *ROS Tutorials Understanding Nodes*. Available at http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes.

[75] ROS.org. *ROS Tutorials Understanding Topics*. Available at http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics.

[76] ROS.org. *ROS Tutorials WritingPublisherSubscriber(python)*. Available at http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29.

[77] ROS.org. *The Stage Robot Simulator*. Available at http://rtv.github.io/Stage/.

[78] ROS.org. *Turtlesim-Tutorial-Moving in a Straight Line*. Available at http://wiki.ros.org/turtlesim/Tutorials/Moving%20in%20a%20Straight%20Line.

[79] Matija Rossi. Ros package for distributed control of networks of dynamical systems. Master's thesis, University of Zagreb, 2014.

[80] Alessandro Santoni. A ros-based workspace control and trajectory planner for a seven degrees of freedom robotic arm. Master's thesis, Alma Mater Studiorum Università di Bologna, 2016.

[81] Daniel Serrano. Introduction to ros – robot operating system –.

[82] Ahmed shamim hassan. Observer vs pub-sub pattern, 10 2017.

[83] Kenta Takaya, Toshinori Asai, Valeri Kroumov, and Florentin Smarandache. Simulation environment for mobile robots testing using ros and gazebo, 10 2016.

[84] M Taufiqqurohman and N Sari. Odometry method and rotary encoder for wheeled soccer robot. *IOP Conference Series: Materials Science and Engineering*, 407:012103, 09 2018.

[85] K. Verbiest, S. A. Berrabah, and E. Colon. Autonomous frontier based exploration for mobile robots. In Honghai Liu, Naoyuki Kubota, Xiangyang Zhu, and Rüdiger Dillmann, editors, *Intelligent Robotics and Applications*, pages 3–13, Cham, 2015. Springer International Publishing.

[86] Wikipedia. Andrew ng. Available at https://en.wikipedia.org/wiki/Andrew_Ng.

[87] Wikipedia. md5sum. Available at https://en.wikipedia.org/wiki/Md5sum.

[88] Wikipedia. ping (networking utility). Available at https://en.wikipedia.org/wiki/Ping_(networking_utility).

[89] Wikipedia. Player project. Available at https://en.wikipedia.org/wiki/Player_Project.

[90] Wikipedia. Willow garage. Available at https://en.wikipedia.org/wiki/Willow_Garage.

[91] Keenan Wyrobek. The origin story of ros, the linux of robotics. Available at https://en.wikipedia.org/wiki/Willow_Garage.

[92] Aleksandar Zivkovic. Development of autonomous driving using robot operating system. Master's thesis, Universidad Politécnica de Madrid, 2018.