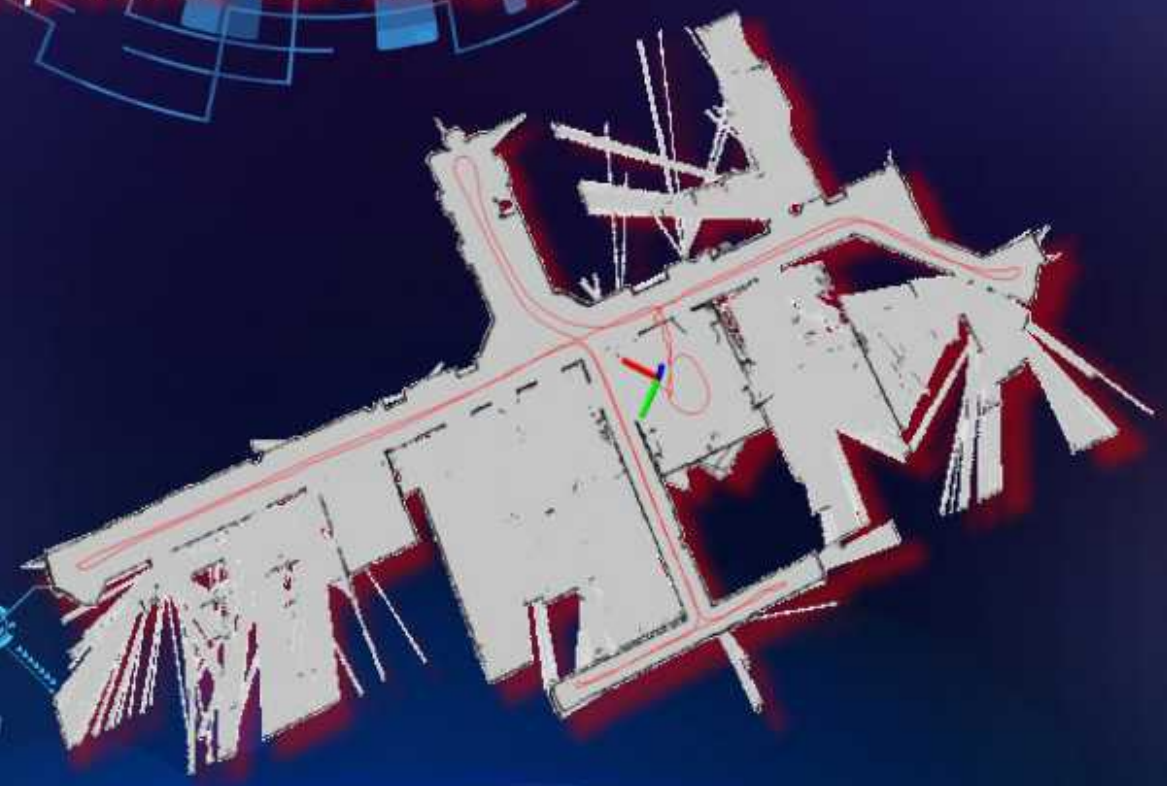
ROS programming with Python

Specific for Kinetic



Alejandro Alonso Puig

ROS

ROS programming with Python

Specific for Kinetic

First Edition
January 2019

Alejandro Alonso Puig
Linkedin: https://www.linkedin.com/in/alejandroalonsopuig/
Blog: http://automacomp.blogspot.com/
Mail: aalonsopuig@gmail.com

Dedicated to my wife, Blanca and my two little kids, Jorge and Daniel.
With gratitude to their patience and encourage to prepare this book with passion.

1 About Alejandro Alonso Puig

Alejandro Alonso Puig is Vicepresident of HISPAROB, the Professional Spanish Robotics Platform[1] and Software Engineering Director at ASTI Mobile Robotics in Spain[2].

He was Chief Technology Officer at Infinium Robotics in Singapore[3], Aerial Robotics Systems Manager at IXION Industry and Aerospace, CEO and CTO at Quark Robotics and also worked in IBM and Honeywell Group, managing IT services.

He was founding President of ARDE, the Spanish Robotics Association, Professional Lecturer in areas of Robotics and Innovation at Thinking Heads[4] and Associate Professor at IE Business School[5]

He has been always passionate about robotics and automation, building and programming many robots of different types. He publish about his robots and thoughts in his blog[6]

Linkedin: https://www.linkedin.com/in/alejandroalonsopuig/
Blog: http://automacomp.blogspot.com/
Mail: aalonsopuig@gmail.com

[1] https://www.hisparob.es/

[2] https://asti.es

[3] https://www.infiniumrobotics.com/

[4] https://www.thinkingheads.com/en/

[5] https://www.ie.edu/

[6] http://automacomp.blogspot.com/

2 Motivation

I like writing notes that help me remember things, concepts and structures. I use this notes also for helping people to learn. This is the case of this book titled “ROS programming with Python”, based on many concepts and experiences from me and from others, that may help you to learn ROS. This is why I share it with you. To help you learn this incredible framework.

This book is not supposed to teach you about ROS for beginners, but to be a compendium of supporting notes for the daily usage of ROS. It explores ROS usage on the ROS Kinetic distro from Python perspective, using a Turtlebot robotics base as well as simulations based on Gazebo.

But why ROS Kinetic and not the latest one? Really because from my point of view ROS Kinetic is probably the long term edition (LTE) better supported and with more libraries published, so it is better in terms of learning and testing concepts.

I assume you have ROS kinetic in a computer. If not, you could see on Appendix section how to install it on a computer with Ubuntu 16.04 or on a Raspberry Pi 3B

3 Introduction to ROS

It was not so far in time when I used to program robots using assembler language, a sort of machine code extremely closed to the hardware of the machine. In those times programming a robot for complex functions, like navigating autonomously from one room to another was a difficult and time consuming project, that may take months for a team of people.

Robotics is evolving very fast as well as the open source movement. A really interesting outcome of both evolutions is ROS (Robot Operating System). ROS is an open-source, meta-operating system for robots, is a robotics middleware (i.e. collection of software frameworks for robot software development)[1]. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers[2].

Nowadays it is possible to program a proof of concept for a robotics system in few days, thanks to ROS and all the open source, already available tools. Its integration with Gazebo Open Source Simulator is also a great help on the design and test of algorithms and behaviors.

Just as an example, let me share with you a development I did recently, using ROS. It took me about 10 hours to write on my own an application for a robot to work as a messenger, able to go to any room or location in a house or an office to deliver goods. The function of the robot is very straightforward: You just put something on the tray of the robot and indicate in the touch screen where to deliver. Then the robot, using a 360 degrees laser scanner for detection of the environment and SLAM navigation technology, finds a trajectory to the required location. This development would have taken easily 6 months of work for a team of 5 experienced people 10 years ago. Now you could have it in 10 hours of only one developer and with a hardware technology (robot platform, lidar, embedded computer) that costs in total less than 1,500 us$.

On Appendix section you could find all the documentation about the development of this solution.

Image 1. Messenger Robot navigating at home. Navigation map at the top left corner

But ROS is not only for developing a proof of concept, but real products for consumer and industrial market. In fact, there are may companies producing and selling robot with ROS, like ClearPath, Fetch Robotics, PAL Robotics, Robotnik and much more. I will talk more about it throughout this book.

Thanks for the Open Source philosophy, thanks for the developers of Linux, the Open Source Operating System in which ROS works, thanks for all the hard work done by Willow Garage team for the development they did of ROS and thanks to the incredible community of developers that share their knowledge and code for the Robotics evolution. This is the key factor of this time. We all are working to make the Robotics technology evolve, by sharing, for the art of sharing.

4 Some interesting sources for learning ROS

Apart form the Bibliography section at the end of the book, I would recommend you the following sources of information for learning ROS:

a) Oficial ROS Tutorial Website provided by OSRF is very comprehensive and it is available in multiple languages. It includes details for ROS installation, documentation of ROS, ROS courses&events, etc. and it's completely free. You just need to follow the ROS tutorials provided on ROS Wiki page, and get started: http://wiki.ros.org/ROS/Tutorials

b) Integrated ROS learning platform – Robot Ignite Academy. Interesting online courses and a full online ROS IDE and simulation environment, so you do not need to have anything installed to learn. Just a browser: http://www.theconstructsim.com/

c) Book: Programming Robots with ROS. Morgan Quigley, Brian Gerkey, and William D. Smart. Published by O'Reilly Media, Inc.

d) Book: ROS in 5 days. The Construct. http://www.theconstructsim.com/ros-in-5-days-book-page/

e) Books from Lentin Joseph: http://www.lentinjoseph.com

f) Roboware Studio. A very good IDE to write code for ROS. Open Source. http://www.roboware.me/

g) ROS video tutorial course provided by Dr. Anis Koubaa from Prince Sultan University, is a great starting point to learn ROS. The course combines a guided tutorial, different examples, and exercises with increasing level of difficulty along with an autonomous robot: https://www.youtube.com/watch?v=xgLETnSMMYA&list=PLSzYQGCXRW1HLWHdJ7ehZPA-nn7R9UKPa

5 Structure of ROS

ROS provides a structure for messaging between Nodes (processes). This messaging system uses Topics, that are like pipes that provide a channel of data between nodes. Nodes use Topics to publish information for other nodes in order to communicate. Topics implement a publish/subscribe communication mechanism so nodes could publish to a topic, be subscribed to a topic or both. Topics handle information through Messages.

Also ROS provides the use of Services. Services are synchronous remote procedure calls. The server (which provides the service) specifies a callback to deal with the service request, and advertises the service.

While services are ok for simple interactions like querying status and managing configuration, they are not adequate when you need to initiate a long-running task.

ROS also provides Actions. When you call an Action, you are calling a functionality that another node is providing. Just the same thing as for Services.

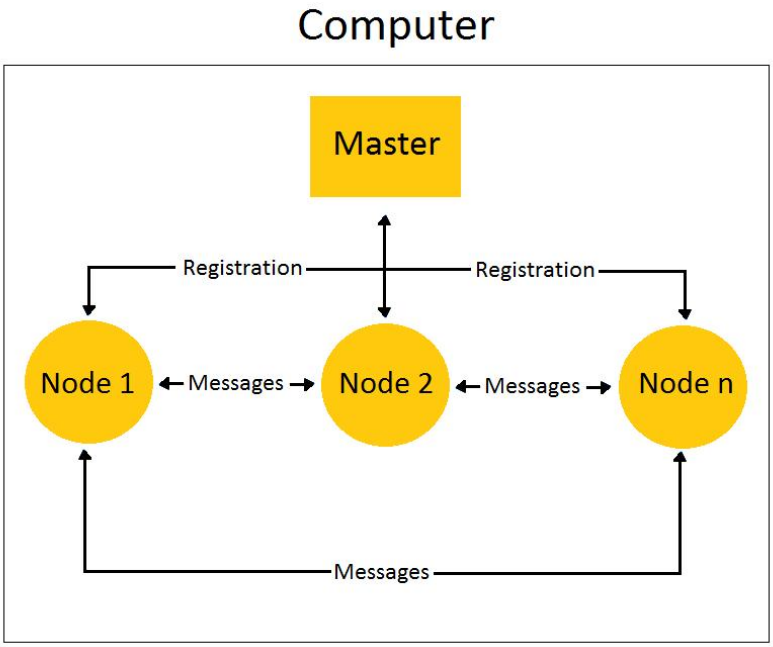


Image 2. Basic concept of ROS

6 Roscore

Roscore is the main process that manages all the ROS system. To run it, introduce '*roscore*' in a terminal on a computer with ROS installed.

6.1 Connection Information

Roscore gives a Service that provides connection information to nodes, so they can transmit messages peer to peer. Nodes connect to Roscore (Master) at startup to register details of the message streams it publishes and the streams it wishes to subscribe.

6.2 Parameter Server

Roscore provides a Parameter Server used by nodes for configuration.

rosparam set <param> <value>	*set a value and create the parameter if it doesn't exist*
rosparam get <param>	*get value of parameter*
rosparam load <file> <param>	*load parameters from file .yaml*
rosparam dump <file> <param>	*dump parameters to file .yaml*
rosparam delete <param>	*delete parameter*
rosparam list	*gets a list of parameters*

example:

rosparam get rosdistro	gets the ROS distro. Result: kinetic...

See also the section Debugging Tools to learn about *rqt_reconfigure.*

In a Launch file, <*rosparam*> tag enables the use of *rosparam YAML files* for loading and dumping parameters from the ROS Parameter Server. It can also be used to remove parameters. The <*rosparam*> tag can be put inside of a <node> tag, in which case the parameter is treated like a private name.

Example:

rosparam command="load" file="$(find pkg-name)/example.yaml" /

7 Topics

A Topic is a message stream[3]. Nodes use Topics to publish information that could be used by other nodes. Topics implement a publish/subscribe communication mechanism.

An example of usage of a Topic is a Lidar or other sensors sending data at 30hz for the navigation system.

Some commands to use in a terminal:

rostopic list	shows the existing list of topics
rostopic list | grep <topic name>	shows the topic if it exists
rostopic info <topic name>	info about topic (Type of message, publisher (name, address and port) and subscribers(name, address and port))
rostopic echo <topic> -n 5	shows the last 5 values published into a topic
rostopic echo <topic>	shows constantly the value of the topic
rostopic -h	shows options of rostopic usage
rostopic pub <topic> <message type> <value>	publishes a message with a value in the Topic.
rostopic bw	display bandwidth used by topic
rostopic find	find topics by type
rostopic hz	display publishing rate of topic
rostopic type	print topic type

Examples:

rostopic pub /count std_msgs/Int32 5	Sets a value of 5 in /count
rostopic pub -r 10 /count std_msgs/Int32 5	Value is sent at a rate of 10Hz in /count
rostopic echo /counter	Shows the value in the count topic

7.1 Topic messages

7.1.1 Overview

Topics handle information through Messages. These Messages can be of many types provided by ROS or even personalized[4].

rosmsg show <message type>	shows the variables and types of them used in the <message type>

Example:

rosmsg show std_msgs/int32

The result is 'int32 data'. It means that message 'std_msgs/int32' has only one variable named 'data' of type 'int32'

rosmsg show	Show message description
rosmsg list	List all messages
rosmsg md5	Display message md5sum
rosmsg package	List messages in a package
rosmsg packages	List packages that contain messages

7.1.2 Common Topics

Some common topics used in ROS:

7.1.2.1 /cmd_vel

Used to control the movement of the robot. We could publish to it in our python program using the Twist message by including:

cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)

We can get more details of the message definition in the next section “geometry_msgs package”

In the case of Turtlebot, both for Gazebo and the real robot, the topic used for this purpose is *cmd_vel_mux/input/navi*. Therefore we could keep *cmd_vel* in our program and use a remapping. Example of remapping calling a python program:

./anyprogram.py cmd_vel:=cmd_vel_mux/input/navi

7.1.2.2 /scan

This topic represents output from a laser scanner that is not providing multiple returns per beam. This topic is not present for multi-echo laserscanners in multi-echo modes. It uses the message type *sensor_msgs/LaserScan*. You could see more details about it in following sections.[5]

7.2 Standard message types

ROS offers a rich set of built-in message types.

7.2.1 std_msgs package

The **std_msgs package** defines the primitive types:

Primitive Type	Serialization	C++	Python2	Python3
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	int
uint64	unsigned 64-bit int	uint64_t	long	int

float32	32-bit IEEE float	float	float	
float64	64-bit IEEE float	double	float	
string	ascii string (4)	std::string	str	bytes
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time	
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration	

7.2.2 geometry_msgs package

But there are other messages types in other available packages. For example the **geometry_msgs package** provides messages for common geometric primitives such as points, vectors, and poses. (full info here: http://wiki.ros.org/geometry_msgs).

One of the messages provided by this package is Twist. Therefore, if we want to use this message type in our program, we need to import it including the following line in our Python code:

from geometry_msgs.msg import Twist

Note: We should not forget to include in package.xml the line: <run_depend>geometry_msgs</run_depend>

This message is used to express velocity in free space broken into its linear and angular parts. So the result of

rosmsg show geometry_msgs/Twist

is a double vector describing the 6DOF:

geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z

ROS uses a right-hand convention for orienting the coordinate axes. The index and middle fingers point along the positive x and y axes and the thumb points in the direction of the positive z axis. The direction of a rotation about an axis is defined by the right-hand rule: If you point your thumb in the positive direction of any axis, your fingers curve in the direction of a positive rotation.

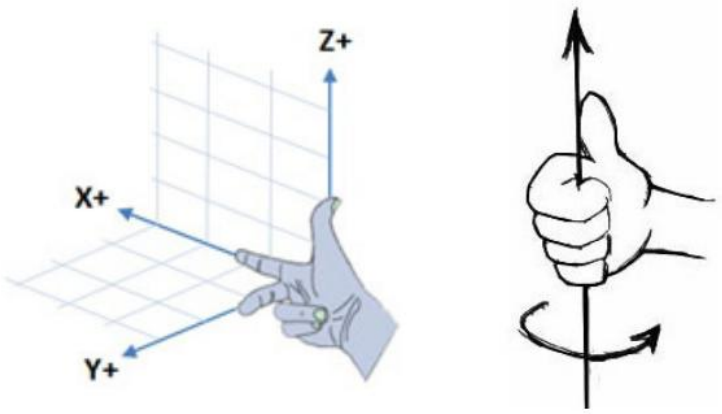


Image 3. The direction of a rotation about an axis is defined by the right-hand rule

Depending on the robot, we may use only a few of them. In the case of Turtlebot/Kobuki, the robot uses a differential drive plugin to move. That is, the robot can only move linearly in the 'x' axis, or rotationally in the angular 'z' axis. This means that the only values that you need to fill in the Twist message are the linear x and the angular z.

ROS uses the metric system so that linear velocities are always specified in meters per second (m/s) and angular velocities are given in radians per second (rad/s). A linear velocity of 0.5 m/s is actually quite fast for an indoor robot (about 1.1 mph) while an angular speed of 1.0 rad/s is equivalent to about one rotation in 6 seconds or 10 RPM. When in doubt, start slowly and gradually increase speed. For an indoor robot in a cluttered area, tend to keep the maximum linear speed at or below 0.3 m/s.

Example of Terminal command: Robot will move in a clockwise circle by publishing the following Twist message:

rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.1, y: 0, z: 0}, angular: {x: 0, y: 0, z: -0.5}}'

We use the -r parameter to publish the Twist message continually at 10Hz. Some robots like the TurtleBot require the movement command to be continually published or the robot will stop as a safety feature. To stop the robot from rotating, type Ctrl-C

Example of Python code:

```
move_cmd = Twist()          # Twist is a datatype for velocity
move_cmd.linear.x = 0.2     # let's go forward at 0.2 m/s
move_cmd.angular.z = 0      # let's turn at 0 radians/s
```

7.2.3 sensor_msgs/LaserScan Message

This message is used for publishing 2D laser range date. Its structure is the following:

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

Header header               # timestamp in the header is the acquisition time of the first ray in the scan.

                            # in frame frame_id, angles are measured around the positive Z axis (counterclockwise, if Z is up)
                            # with zero angle being forward along the x axis

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

float32 time_increment      # time between measurements [seconds] - if your scanner is moving, this will be used in interpolating position
                            # of 3d points
float32 scan_time           # time between scans [seconds]

float32 range_min           # minimum range value [m]
float32 range_max           # maximum range value [m]

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

Ranges is an array of values with distances from the laser to the obstacles. The amount of data stored in this array (indexes) is (*angle_max* - *angle_min*) / *angle_increment* that is also the value of *len(ranges)*

Therefore the front of the sensor would be the middle value for the index *ranges*[index]

Usually the angle considered 0 radians is the front of the sensor. See sensor documentation just in case.

Following function gives the distance value in meters at a specific angle. *zero_index* is the index with the beam pointing to the front of the robot. In this case we considered a Hokuyo Lidar, with the middle of the angle of the beam being pointing at the front of the robot. If the Lidar is installed in other position, this should be taken in account in the calculation of *zero_index* variable. This function is called from the callback.

```
def scan_distance(scan_data, degrees_angle):
# get the distance measured by the laser in the angle given in degrees
# being the front of the laser the angle 0
# scan_data is of type sensor_msgs/LaserScan

radians_angle=radians(degrees_angle)                # don't forget 'from math import radians'
zero_index=len(scan_data.ranges)/2                  #index corresponding to 0 degree
index = int(zero_index + radians_angle/scan_data.angle_increment)
if index>len(scan_data.ranges): index=len(scan_data.ranges)-1
if index<0: index=0
distance = scan_data.ranges[index]
return distance
```

This command get the minimum distance to an obstacle in the full range of the lidar:

```
min_obstacle = min(scan_data .ranges)
```

7.2.4 Other common messages

See[6]: http://wiki.ros.org/common_msgs

7.3 Defining Custom messages

ROS already has a rich set of message types, and you should use one of these if you can. So, before you go and create a new message type, you should use *rosmsg* to see if there is already something there that you can use instead. Anyway, here are some tips on how to create new messages[7].

Let's see with an example of specific message for Drones control on how to create personalized ROS messages:

1. Create personalized message directory (msg) inside the package
2. Inside msg, create file (xx.msg). It should contain the message structure, like

```
string command
float32 height
int32 yaw_offset
```

3. Modify CMakeList.txt adding:

```
find_package(catkin REQUIRED COMPONENTS
message_generation
std_msgs
rospy
)

add_message_files(FILES
uav_command.msg (your message definition filename)
)

generate_messages(DEPENDENCIES
std_msgs
)

catkin_package(
CATKIN_DEPENDS
message_runtime
std_msgs
)
```

4. Modify package.xml adding:

```
<build_depend>std_msgs</build_depend>
<run_depend>std_msgs</run_depend>
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

5. Compile

roscd
cd ..
catkin_make <package>
verify the message exist from now executing rosmsg list. the message should exist with name: /<package name>/<msg def file name>

6. Use in code

To use in code you need to add in your code:

```
from <package name>.msg import *
```

(where <package name> is the name of the package that has the msg file, so the actual package)
then I set a publisher in the normal way:

```
pub = rospy.Publisher('/<package name>/<msg def file name>', <msg def file name>, queue_size=1)
```

where <package name> is the same mentioned before and
<msg def file name> is the name of the msg definition file created under the msg folder, but without .msg extension.

Then I use it:

```
pubdata = <msg def file name>()
```

example:

```
from wh_nav_v2.msg import *

cmd_drone = rospy.Publisher('/wh_nav_v2/uav_command', uav_command, queue_size=1)
rate = rospy.Rate(20)
cmd_to_drone = uav_command()

cmd_to_drone.command = 'TAKEOFF'
cmd_to_drone.height = 2.0
cmd_to_drone.yaw_offset = 0
cmd_drone.publish(cmd_to_drone)
```

7.4 Latched Topics

If you're not subscribed to a topic when a message goes out on it, you will miss it and you will have to wait for the next one. This is not a problem if the publisher sends out messages frequently. However, there are cases where sending out frequent messages is a bad idea. For example, the map_server node advertises a map (of type *nav_msgs/OccupancyGrid*) on the map topic. This represents a map of the world that the robot can use to determine where it is. Often, this map never changes and is published only once, when the map_server loads it from disk. However, this means if another node needs the map, but starts up after map_server publishes it, it will never get the message. We could periodically publish the map, but we don't want to publish the message more often than we have to, since it's typically huge.

Latched topics offer a simple solution to this problem. If a topic is marked as latched when it is advertised, subscribers automatically get the last message sent when they subscribe to the topic. In our map_server example, this means that we only need to mark it as latched and publish it once. Topics can be marked as latched with the optional latched argument:

```
pub = rospy.Publisher(' map', nav_msgs/ OccupancyGrid, latched = True)
```

From Terminal, rostopic pub has a -l option for latching. See *rostopic pub -h*

7.5 Topics remapping

When running a node that is subscribed to a topic, which name is different than the topic really publishing, then we do a remapping. This is quite common with the lidar data, being sometimes publish in */scan* topic and some other in */laserscan* topic.

Only *rosrun* and not *roslaunch* accept remapping. Example where we remap */laserscan* to */scan*. */laserscan* is the real topic were we are publishing, but */scan* is the topic to which slam_gmapping[8] is subscribed:

rosrun gmapping slam_gmapping /scan:=/laserscan

8 Catkin workspace

Catkin is the ROS build system. catkin comprises a set of CMake macros and custom Python scripts to provide extra functionality on top of the normal CMake workflow. CMake is a commonly used open source build system.

A workspace is simply an area, a set of directories in which a related set of ROS code lives. You can have multiple ROS workspaces, but you can only work in one of them at any one time.

Catkin is a specific ROS workspace[9], a directory where our ROS packages must reside in order to be usable by ROS. Called also Catkin_ws.

To move to this workspace:

roscd
cd..

This folder contains three other folders:

build
devel
src: contains all packages created

9 Packages

9.1 Overview

Software in ROS is organized in packages. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages it to provide this useful functionality in an easy-to-consume manner so that software can be easily reused. In general, ROS packages follow a "Goldilocks" principle: enough functionality to be useful, but not too much that the package is heavyweight and difficult to use from other software[10].

Packages are organized with the following structure:

a) Launch folder: Contains launch files. Their extension is '.launch'. Launch files are XML files that describe a collection of nodes along with their topic remappings and parameters. Inside these files there is a node tag. Each < node > tag includes attributes declaring the ROS graph name of the node, the package in which it can be found, and the type of node, which is simply the filename of the executable program.

Example of Launch file content:

```
< launch >
< node name =" talker" pkg =" rospy_tutorials"
type =" talker.py" output =" screen" />
< node name =" listener" pkg =" rospy_tutorials"
type =" listener.py" output =" screen" />
</ launch >
```

In case of using a Cpp node:

```
<node pkg ="package_where_cpp_is"
type="name_of_binary_after_compiling_cpp"
name="name_of_the_node_initialised_in_cpp"
output="screen">
</node>
```

b) src folder: source files (cpp, python)
c) cmakelists.txt: List of cmake rules for compilation
d) package.xml: Package information and dependencies

rosrun <package> <python executable> allows to run a ROS program without having to create a launch file to launch it.

An example of running an executable that allows to input movement commands through the keyboard is the following:

rosrun teleop_twist_keyboard teleop_twist_keyboard.py

Other option to call a python program is going to its folder and calling it like in this example:

./teleop_twist_keyboard.py

Although rosrun is great for starting single ROS nodes during debugging sessions, most robot systems end up consisting of tens or hundreds of nodes, all running at the same time. Since it wouldn't be practical to call rosrun on each of these nodes, ROS includes a tool for starting collections of nodes, called roslaunch.

roslaunch <package-name> <launch-file>

example:

roslaunch turtlebot_teleop keyboard_teleop.launch

Basically, this starts a node that listens to the keyboard activity and publishes messages to the topic */cmd_vel*. The robot listens to this topic and drives to the desired direction.

Ctrl-C is a common way to force programs to exit on the Linux/Unix command line, and *roslaunch* follows this convention by closing its collection of launched nodes and then finally exiting roslaunch itself when Ctrl-C is typed into its console.

roscd <package name> takes me to the directory where the package is

rospak list	gives a list of all packages in our ROS System
rospack list | grep my_package	shows only the route to 'my_package'

If package is not found: source ../../devel/setup.bash

9.2 Call other packages from our package

Sometimes it is necessary to call other packages before starting ours. For example running the Package that start running the Lidar node, before our navigation package starts. The way to do this is to add a line like this at the beginning of the launch file, just after the <launch> first line:

```
<include file="$(find <package name>)/launch/<launch file name>"/>
```

for example:

```
<include file="$(find iri_wam_reproduce_trajectory)/launch/start_service.launch"/>
```

9.3 Create a package

Packages are easy to create by hand or with tools like catkin_create_pkg. A ROS package is simply a directory descended from ROS_PACKAGE_PATH (see ROS Environment Variables) that has a package.xml file in it. Packages are the most atomic unit of build and the unit of release. This means that a package is the smallest individual thing you can build in ROS and it is the way software is bundled for release (meaning, for example, there is one debian package for each ROS package), respectively.

Steps to create a package as follows:

1. Go to Catkin_ws:

   Every time we want to create a package, we need to be in the Catkin_ws/src directory.

   cd ~/catkin_ws/src

2. Create a package:

catkin_create_pkg <package> <dependencies>

examples:

catkin_create_pkg my_package rospy
catkin_create_pkg move_robot geometry_msgs rospy

after doing this, the package exists and *rospack list* will show it

3. Go to the package directory:

roscd <package name>

4. Create a Python program:

cd src
touch <program name>
edit it
chmod +x <program name> (execution rights)

5. Create Launch directory and file

cd ..
mkdir launch
cd launch
touch <launch file>
edit it ----> Something like this should be ok to start:

```
<launch>
<!-- My Package launch file -->
<node pkg="<package name>" type="<program name>" name="<node name>"  output="screen">
```

```
</node>
</launch>
```

6. Compile a package:

cd ~/catkin_ws
catkin_make --only-pkg-with-deps <package>

7. Execute:

roslaunch <package-name> <launch-file>

9.4 Calling a package with parameters

Let's use an example on how to call a package with parameters:

```
if rospy.has_param('~linear_scale'):
        g_vel_scales[1] = rospy.get_param('~linear_scale')
else:
        rospy.logwarn("linear scale not provided; using %.1f" %\
        g_vel_scales[1])
if rospy.has_param('~angular_scale'):
        g_vel_scales[0] = rospy.get_param('~angular_scale')
else:
        rospy.logwarn("angular scale not provided; using %.1f" %\
        g_vel_scales[0])
```

10 Nodes

A node is a process that performs tasks. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes. For example, one node controls a laser range-finder, one Node controls the robot's wheel motors, one node performs localization, one node performs path planning, one node provides a graphical view of the system, and so on[11].

A Node can provide a functionality to other nodes through a Service call, so they can call the service when they need such functionality. Example, processing a computational calculation.

rosnode list	tells nodes actually running
rosnode info /<node name>	gives information about the publications, subscriptions, services given, and other

rqt_graph is a ROS graph visualizer. This will bring up a display of the connections between nodes. This renderings will not autorefresh, but you can click the refresh icon in the upper-left corner of the rqt_graph window when you add a node to or remove one from the ROS graph by terminating (e.g., pressing Ctrl-C) or running (via rosrun) its program, and the graph will be redrawn to represent the current state of the system.

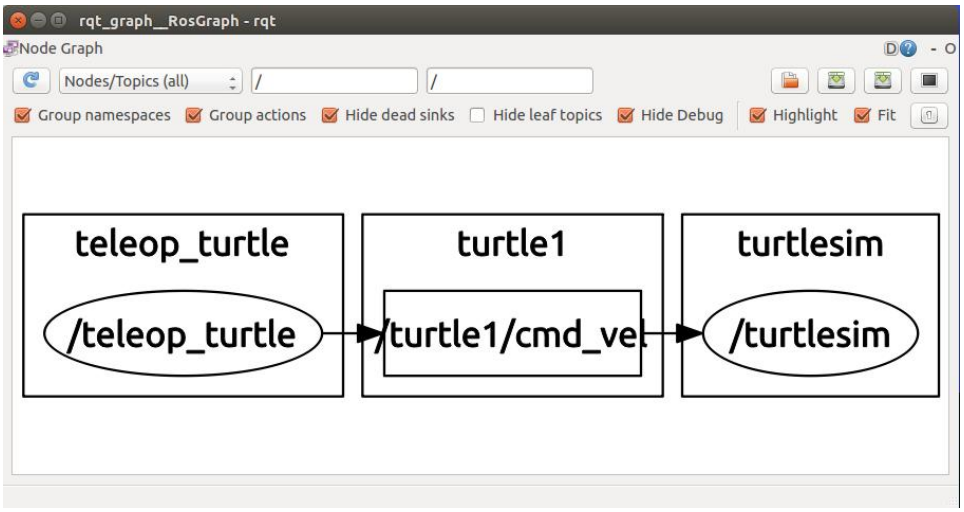


Image 4. Example of rqt_graph[12]

Typically Nodes are POSIX processes and connections between nodes are TCP connections.

Before nodes start to transmit data over topics, they must first announce, or advertise, both the topic name and the types of messages that are going to be sent. Then they can start to send, or publish, the actual data on the topic. Nodes that want to receive messages on a topic can subscribe to that topic by making a request to roscore. After subscribing, all messages on the topic are delivered to the node that made the request. One of the main advantages of using ROS is that all the messy details of setting up the necessary connections when nodes advertise or subscribe to topics is handled by the underlying communication mechanism.

10.1 Publishers

A Publisher is a node that keeps publishing a message into a Topic.

Example of publisher: a Node called 'Topic_publisher'. It publishes at a frequency of 10Hz a increasing number 'count' on the '/counter' topic.

```
#! /usr/ bin/ env python                    #It is a Python file and should be passed to Python interpreter

import rospy                                # Import the Python library for ROS
from std_msgs.msg import Int32              # Import the Int32 message from the std_msgs package
rospy.init_node('topic_publisher')          # Initiate a node named 'topic_publisher'
pub = rospy.Publisher('counter', Int32)     # Create a Publisher object, that publish on the /counter topic  messages of type Int32
rate = rospy.Rate(10)                       # Set a publish rate of 10 Hz
count = Int32()                             # Creates var of type msg Int32
count.data = 0                              # Initialize 'count'

while not rospy.is_shutdown():              # Create a loop that will go until someone stops the program execution
pub.publish(count)                          # Publish the message within the 'count' variable
count.data + = 1                            # Increment 'count' variable
rate.sleep()                                # Make sure the publish rate maintains at 2 Hz
```

More details on how to write a Publisher[13]:
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

10.2 Subscriber

A Subscriber is a node that reads information from a topic.

Example of a Subscriber:

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

import rospy
from std_msgs.msg import Int32

def callback( msg):                          # Define a function called 'callback' that receives a parameter named 'msg'
print msg.data                               # Print the value 'data' inside the 'msg' parameter

rospy.init_node(' topic_subscriber')         # Initiate a node called 'topic_subscriber'
sub = rospy.Subscriber(' counter', Int32, callback)  # Create a subscriber object  that will listen to the /counter  topic and will call
# the 'callback' function each time it reads something from  the topic
rospy.spin()                                 # Create a loop that will keep the program in execution
```

More details on how to write a Subscriber:
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

10.3 Mixing Publishers and Subscribers

One of the most common things nodes in ROS do is to transform data by performing computations on it. For example, a node might subscribe to a topic containing camera images, identify objects in those images, and publish the positions of those objects in another topic.

Example. increase10.py :

The subscriber and publisher are set up as before, but now we're going to publish data in the callback, rather than periodically. The idea behind this is that we only want to publish when we have new data coming in, since the purpose of this node is to transform data (in this case, increasing by 10 the number that comes in on the subscribed topic).

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

import rospy from std_msgs.msg import Int32
rospy.init_node('Increase10')

def callback(msg):
increased = Int32()
increased.data = msg.data +10
pub.publish(increased)

sub = rospy.Subscriber('number', Int32, callback)
pub = rospy.Publisher('increased', Int32)

rospy.spin()
```

11 Services

11.1 Overview

The publish/subscribe model is a very flexible communication paradigm, but its many-to-many one-way transport is not appropriate for RPC request/reply interactions, which are often required in a distributed system. Request/reply is done via a Service, which is defined by a pair of messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply. Client libraries usually present this interaction to the programmer as if it were a remote procedure call[14].

Services are another way to pass data between nodes in ROS. Services are just synchronous remote procedure calls; they allow one node to call a function that executes in another node. We define the inputs and outputs of this function similarly to the way we define new message types. The server (which provides the service) specifies a callback to deal with the service request, and advertises the service. The client (which calls the service) then accesses this service through a local proxy (http://wiki.ros.org/proxy).

Services aren't always the best fit. While services are handy for simple get/set interactions like querying status and managing configuration, they don't work well when you need to initiate a long running task. When a ROS program calls a service, it could not continue until it receives a result from the service.

rosservice list	List of services running (active)
rosservice type /<name of service>	print service type as <package name>/<service message>
rossrv list	All available services as <package name>/<service message>
rossrv packages	All packages offering services
rossrv package <package name>	services offered by a single package
rosservice info /<name of service>	gives information on the node that provides the service, the URI (ip:port), the type of message used by the service and the arguments that the service takes when called.
rosservice call /<name of service> <parameters>	calls a service from the console
rosservice show <package	shows the structure of the service message

name>/<service message>

11.2 Service messages

Service Definition Files have extension '.srv' and are inside the srv directory of the package, instead of an msg directory as with Topic messages.

Example that shows the srv message files for gazebo:

roscd gazebo_msgs
ls srv

As well as with Topics, it is possible to generate personalized service messages.

In order to know the structure of the service message used by the service you have to execute first **rosservice info /<name of service>**, to get the type of service message. Then, you can explore the structure of that service message with the following command:

rossrv show <name_of_the_package>/<Name_of_Service_message>

Example:

rossrv show gazebo_msgs/DeleteModel

Result:

string model_name
---
bool success
string status_message

This is the same as doing:

roscd gazebo_msg

```
cd srv
cat DeleteModel.srv
```

Service messages have TWO parts:

REQUEST

RESPONSE

In the case of the DeleteModel service, REQUEST contains a string called model_name and RESPONSE is composed of a boolean named success, and a string named status_message.

11.3 Implementing a Service

There are several steps to create a new Service:

11.3.1 Service call inputs and outputs definition

1. Create a package: If it doesn't already exist, create it following the steps described in
2. Create a service-definition file, with extension '.srv' and are inside the srv directory of the package, as explained previously. Therefore we need to have a clear idea of inputs and outputs of our service and then create the directory srv and definition file.
3. Make sure that the find_package() call in CMakeLists.txt contains message_generation and std_msg (in addition to any other packages that are already there). This will allow catkin_make to create later the code and class definitions that we will actually use when interacting with the service.

```
find_package(catkin REQUIRED COMPONENTS
# other packages are already listed here
message_generation # Add message_generation here, after the other packages
std_msgs
)
```

4. We need to tell catkin which service-definition files we want compiled, using the add_service_files() call in CMakeLists.txt:

```
add_service_files(
FILES
<service-definition file name>.srv
)
```

5. Now we must make sure that the dependencies for the service-definition file are declared (again in CMakeLists.txt), using the generate_messages() call:

```
generate_messages(
DEPENDENCIES
std_msgs
)
```

6. We also have to make an addition to the package.xml file to reflect the dependencies on both rospy and the message system. This means we need a build dependency on std_msgs, message_generation and a runtime dependency on message_runtime:

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>std_msgs</build_depend>
<run_depend>std_msgs</run_depend>
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
<build_depend>rospy</build_depend>
<run_depend>rospy</run_depend>
```

7. With all of this in place, Run catkin_make:

roscd
cd ..
catkin_make
source devel/setup.bash

Note: source devel/setup.bash. This executes the bash file that sets, among other things, the newly generated messages created with catkin_make. If you don't do this, it might give you a python import error, saying that it doesn't find the Message generated.

This will generate three classes:

<*class name*>
<*class name*>Request, and
<*class name*>Response

Being <*class name*> the same name used for the <*service-definition file*> before. These classes will be used to interact with the service. They are generated in a Python module with the same name as the package, with a .srv extension.

We can verify that the service call definition is what we expect by using the rossrv command:

rossrv show <service-definition file name> (without the .srv extension)

Next, after running the package that provides the service with rosrun:

rosservice list --> verify the service is running
rosservice info <service name> --> This tells us the node that provides the service, where it's running, the type that it uses, and the names of the arguments to the service call.

11.3.2 Create the service code

Like topics, services are a callback-based mechanism. The service provider specifies a callback that will be run when the service call is made, and then waits for requests to come in.

We would create the python code in the .py file of the package.

At the beginning of the python script, after importing rospy, we import the code generated by catkin previously:

```
from <package_name>.srv import <class name>, <class name>Response
```

Example of a Server creation that just prints 'My_callback has been called'. The Service name is '/my service' and this service calls a function called 'my_callback' that prints the mentioned string. The code just creates the service, but should be called when needed. We could call it manually by using the command:

rosservice call /my_service "{}"

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

import rospy
from std_srvs.srv import Empty, EmptyResponse  # you import the service message python classes generated from Empty.srv.
def my_callback( request):
print "My_callback has been called"
return EmptyResponse()                         # the service Response class, in this case EmptyResponse

rospy.init_node('service_client')
my_service = rospy.Service('/ my_service', Empty , my_callback)
# create the Service called my_service with the defined callback
rospy.spin()                                   # maintain the service open.
```

11.4 Service Client

Example of a service client that request (to service /gazebo/delete_model) for the deletion of an object in Gazebo:

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

import rospy
from gazebo_msgs.srv import DeleteModel, DeleteModelRequest # Import service message used by service
import sys

rospy.init_node('service_client')                  # Initialise a ROS node with the name service_client
rospy.wait_for_service('/gazebo/delete_model')     # Wait for service client to be running
```

```
delete_model_service = rospy.ServiceProxy('/gazebo/delete_model', DeleteModel) # Create the connection to the service
kk = DeleteModelRequest()                                                      # Create an object of type DeleteModelRequest
kk.model_name = "bowl_1"                                                       # Fill the variable model_name of this object with the desired value
result = delete_model_service(kk)                                              # Send through connection the name of object to deleted by service
print result                                                                   # Print the result given by the service called
```

12 Actions

12.1 Overview

Actions are a form of asynchronous communication in ROS. Action clients send goal requests to action servers. Action servers send goal feedback and results to action clients[15].

Services aren't always the best fit, either, in particular when the request that's being made is more than a simple instruction of the form "get (or set) the value of X."

While services are handy for simple get/set interactions like querying status and managing configuration, they don't work properly when you need to initiate a long-running task.

ROS actions are the best way to implement interfaces to time-extended, goal-oriented behaviors like goto_position.

When you call an Action, you are calling a functionality that another node is providing. Just the same thing as for Services.

12.2 Action messages

Similar to the request and response of a service, an action uses a goal to initiate a behavior and sends a result when the behavior is complete. But the action further uses feedback to provide updates on the behavior's progress toward the goal and also allows for goals to be canceled. Actions are themselves implemented using topics. An action is essentially a higher-level protocol that specifies how a set of topics (goal, result, feedback, etc.) should be used in combination.

The Node that provides the functionality has to contain an *Action Server*. The *Action Server* allows other nodes to call that action functionality.

The nodes that calls to the functionality has to contain an *Action Client*. The *Action Client* allows a node to connect to the *Action Server* of another Node.

Example of launching an *Action Server* for the ArDrone:

roslaunch ardrone_as action_server.launch

In order to know which *Action Servers* are available on a robot, we must do a:

rostopic list

Every Action Server creates 5 Topics with messages to communicate with it: Goal, Cancel, Status, Result and Feedback.

Example: In the case of ArDrone would create:

/ardrone_action_server/cancel
/ardrone_action_server/feedback
/ardrone_action_server/goal
/ardrone_action_server/result
/ardrone_action_server/status

Once an Action Server is launched, it could be called by sending messages to it.

The message of a Topic is composed by a single part: The information the topic provides.
The message of a Service has two parts: The goal and the response
The message of an Action Server is divided into three parts: The goal, the result and the feedback. Each one can contain more than one variable.

All the action messages used by an Action Server are defined in the Action Directory of the package, in a file with extension '.action'. For example, to see the action messages of ArDrone we do:

roscd ardrone_as/action
cat Ardrone.action

The *Feedback* is a message that the Action Server generates every once in a while to indicate how is the action going, informing the caller the status of the requested action. It is generated while the action is in progress.

Client.Cancel_goal() cancel a *Goal* previously sent to an Action Server prior to its completion.

12.3 Implementing an Action Server

There are several steps to create a new Action:

12.3.1 Defining an Action

1. Create a package: If it doesn't already exist, create it following the steps described in
2. Create an Action definition file. It is to define the goal, result, and feedback message formats in an action definition file, which by convention has the suffix .action. The .action file format is similar to the .srv format used to define services, just with an additional field. And, as with services, each field within an .action file will become its own message. The action file should be placed in a directory called *action* within a ROS package.

   Just like with service-definition files, we use three dashes (---) as the separator between the parts of the definition. While service definitions have two parts (request and response), action definitions have three parts (goal, result, and feedback).

3. Make sure that the find_package() call in CMakeLists.txt contains actionlib_msgs (in addition to any other packages that are already there). This will allow catkin_make to create later the code and class definitions that we will actually use when interacting with the service.

   ```
   find_package(catkin REQUIRED COMPONENTS
   # other packages are already listed here
   actionlib_msgs
   )
   ```

4. We need to tell catkin which action-definition files we want compiled, using the add_action_files() call in CMakeLists.txt:

   ```
   add_action_files(
   FILES
   ```

```
<action-definition file name>.action
)
```

5. Now we must make sure that the dependencies for the action-definition file are declared (again in CMakeLists.txt), using the generate_messages() call:

```
generate_messages(
DEPENDENCIES
actionlib_msgs
std_msgs
)
```

6. Add actionlib_msgs as a dependency for catkin:

```
catkin_package(
        CATKIN_DEPENDS
        rospy
        actionlib_msgs
)
```

7. We also have to make an addition to the package.xml file to reflect the dependencies on both rospy and the message system. This means we need a build/run dependency on **std_msgs** and **actionlib_msgs**:

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>std_msgs</build_depend>
<run_depend>std_msgs</run_depend>
<build_depend>actionlib_msgs</build_depend>
<run_depend>actionlib_msgs</run_depend>
<build_depend>rospy</build_depend>
<run_depend>rospy</run_depend>
```

8. With all of this in place, Run catkin_make:

roscd
cd ..

catkin_make
source devel/setup.bash

Note: source devel/setup.bash. This executes the bash file that sets, among other things, the newly generated messages created with catkin_make. If you don't do this, it might give you a python import error, saying that it doesn't find the Message generated.

This will generate several message definition files:

<*action-definition file name*>Action.msg
<*action-definition file name*>Action-Feedback.msg
<*action-definition file name*>ActionGoal.msg
<*action-definition file name*>ActionResult.msg
<*action-definition file name*>Feedback.msg
<*action-definition file name*>Goal.msg
<*action-definition file name*>Result.msg

These messages are used to implement the action client/server protocol, which, as mentioned previously, is built on top of ROS topics. The generated message definitions are in turn processed by the message generator to produce corresponding class definitions, with same names. They are generated in a Python module with the same name as the package, with a .msg extension.

More information on how to create an action server in Python:
https://index.ros.org/doc/ros2/Tutorials/Actions/Writing-an-Action-Server-Python/

12.3.2 Create the action server code

Like topics and services, actions are a callback-based mechanism, with your code being invoked as a result of receiving messages from another node.

We would create the python code in the .py file of the package.

The easiest way to build an action server is to use the SimpleActionServer class from the actionlib package.

Example: *As a simple example, let's define an action that acts like a timer. We want this timer to count down, signaling us when the specified time has elapsed. Along the way, it should tell us periodically how much time is left. When it's done, it should tell us how much time actually elapsed. (We're building a timer because it's a simple example of an action. In a real robot system, you would use the time support that is built into ROS client libraries, such as rospy.sleep().)*

Action definition file 'Timer.action has format:

```
# This is an action definition file, which has three parts: the goal, the
# result, and the feedback.
#
# Part 1: the goal, to be sent by the client
#
# The amount of time we want to wait
duration time_to_wait
---
# Part 2: the result, to be sent by the server upon completion
#
# How much time we waited
duration time_elapsed
# How many updates we provided along the way
uint32 updates_sent
---
# Part 3: the feedback, to be sent periodically by the server during
# execution.
#
# The amount of time that has elapsed from the start
duration time_elapsed
# The amount of time remaining until we're done
duration time_remaining
```

The .py would be:

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


#First we import the standard Python time
#package, which we'll use for the timer functionality of our server. We also import the
#ROS actionlib package that provides the SimpleActionServer class that we'll be
#using. Finally, we import some of the message classes that were autogenerated from
#our Timer.action file
```

```
import time
import actionlib                                                    #provides SimpleActionServer class
from basics.msg import TimerAction, TimerGoal, TimerResult          #'basic' is the name of the package


def do_timer(goal):
#Next, we define do_timer(), the function that will be invoked when we receive a new
#goal. In this function, we handle the new goal in-place and set a result before returning.
#The type of the goal argument that is passed to do_timer() is TimerGoal, which
#corresponds to the goal part of Timer.action. We save the current time, using the standard
#Python time.time() function, then sleep for the time requested in the goal,
#converting the time_to_wait field from a ROS duration to seconds
start_time = time.time()
time.sleep(goal.time_to_wait.to_sec())


#The next step is to build up the result message, which will be of type TimerResult;
#this corresponds to the result part of Timer.action. We fill in the time_elapsed field
#by subtracting our saved start time from the current time, and converting the result
#to a ROS duration. We set updates_sent to zero, because we didn't send any updates
#along the way
result = TimerResult()
result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
result.updates_sent = 0


#Our final step in the callback is to tell the SimpleActionServer that we successfully
#achieved the goal by calling set_succeeded() and passing it the result. For this simple
#server, we always succeed
server.set_succeeded(result)


#Back in the global scope, we initialize and name our node as usual, then create a SimpleActionServer.
#The first constructor argument for SimpleActionServer is the
#server's name, which will determine the namespace into which its constituent topics
#will be advertised; we'll use timer. The second argument is the type of the action that
#the server will be handling, which in our case is TimerAction. The third argument is
#the goal callback, which is the function do_timer() that we defined earlier. Finally,
#we pass False to disable autostarting the server. Having created the action server, we
#explicitly start() it, then go into the usual ROS spin() loop to wait for goals to
#arrive
rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```

We could run the server with rosrun and then verify if everything is ok.

rostopic list will show if the expected topics (cancel, feedback, goal, result, and status) are present.

12.4 Implementing an Action Client

Calling an action server means sending a message to it by using an *Action Client*. Therefore we need to implement an Action Client.

The following is a self-explanatory example of how to implement an action client that calls the ardrone_action_server and makes it take pictures for 10 seconds.

Action definition file *Ardrone.action* has format:

```
#goal for the drone
int32 nseconds  # the number of seconds the drone will be taking pictures
---
#result
sensor_msgs/CompressedImage[] allPictures # an array containing all the pictures taken along the nseconds
---
#feedback
sensor_msgs/CompressedImage lastImage  # the last image taken
```

The Python code is:

```
#! /usr/bin/env python
import rospy
import time
import actionlib       #provides SimpleActionServer class
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult, ArdroneFeedback

nImage = 1
```

```
# definition of the feedback callback. This will be called when feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
global nImage
print('[Feedback] image n.%d received'%nImage)
nImage += 1

# initializes the action client node
rospy.init_node('drone_action_client')

# create the connection to the action server:
# client = actionlib.SimpleActionClient('/the_action_client_server_name', the_action_server_action_message_python_object)
client = actionlib.SimpleActionClient('/ardrone_action_server', ArdroneAction)

# waits until the action server is up and running
client.wait_for_server()

# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

# sends the goal to the action server, specifying which feedback function
# to call when feedback received
client.send_goal(goal, feedback_cb=feedback_callback)

# wait until the result is obtained
# you can do other stuff here instead of waiting
# and check for status from time to time
# status = client.get_state()
client.wait_for_result()

print('[Result] State: %d'%(client.get_state()))
```

get_state(): When called, it returns an integer that indicates in which state is the action that the SimpleActionClient object is connected to.

0 ==> PENDING
1 ==> ACTIVE
2 ==> DONE
3 ==> WARN
4 ==> ERROR

This allows you to create a while loop that checks if the value returned by get_state() is 2 or higher. If it is not, it means that the action is still in progress, so you can keep doing other things.

More information on how to create an action client in Python:
https://index.ros.org/doc/ros2/Tutorials/Actions/Writing-an-Action-Client-Python/

12.5 The axclient

Axclient is a GUI tool provided by the action lib package that allows to interact with an Action Server in a very easy and visual way. It is called by the following command:

rosrun actionlib axclient.py /<name of action server>

example:

rosrun actionlib axclient.py /ardrone_action_server

13 ROS Environment variables

ROS uses a set of Linux System environment variables in order to work properly.

export | grep ROS shows the environment variables.

These are some of the variables:

ROS_MASTER_URI	Contains the URL and port where the roscore is being executed. Usually the URL is the localhost and the port is 11311
ROS_PACKAGE_PATH	Contains the paths in hardrive where ROS has packages in
ROS_DISTRO	Contains version of ROS. for example "indigo"

Detailed description of ROS Environment Variables[16]: http://wiki.ros.org/ROS/EnvironmentVariables

14 Simulations

There are several common simulators used with ROS. Here we briefly talk about turtlesim and stage and more in deep about Gazebo.

14.1 Turtlesim

A simple way to learn the basics of ROS is to use the turtlesim simulator that is part of the ROS installation. The simulation consists of a graphical window that shows a turtle-shaped robot. The background color for the turtle's world can be changed using the Parameter Server. The turtle can be moved around on the screen by ROS commands or using the keyboard[17].

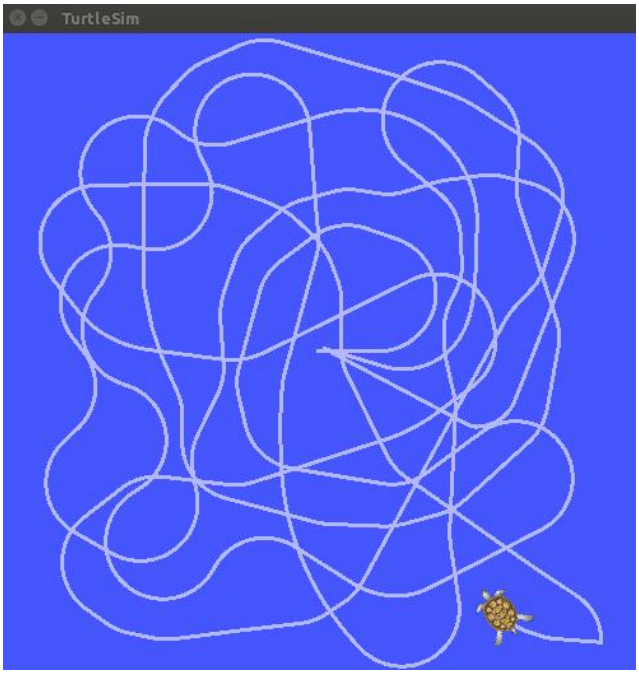


Image 5. Turtlesim

For more information and tutorials of this simulator, visit the following links:

i. Turtlesim – the first ROS robot simulation: https://subscription.packtpub.com/book/hardware_and_creative/9781788479592/1/ch01lvl1sec14/turtlesim-the-first-ros-robot-simulation

ii. Tutorials Using Turtlesim: http://wiki.ros.org/turtlesim/Tutorials

14.2 Stage

Stage is a robot simulator. It provides a virtual world populated by mobile robots and sensors, along with various objects for the robots to sense and manipulate[18].

Stage was designed with multi-agent systems in mind, so it provides fairly simple, computationally cheap models of lots of devices rather than attempting to emulate any device with great fidelity. Stage is intended to be just realistic enough to enable users to move controllers between Stage robots and real robots, while still being fast enough to simulate large populations. We also intend Stage to be comprehensible to undergraduate students, yet sophisticated enough for professional reseachers.

Player also contains several useful 'virtual devices'; including some sensor pre-processing or sensor-integration algorithms that help you to rapidly build powerful robot controllers. These are easy to use with Stage.

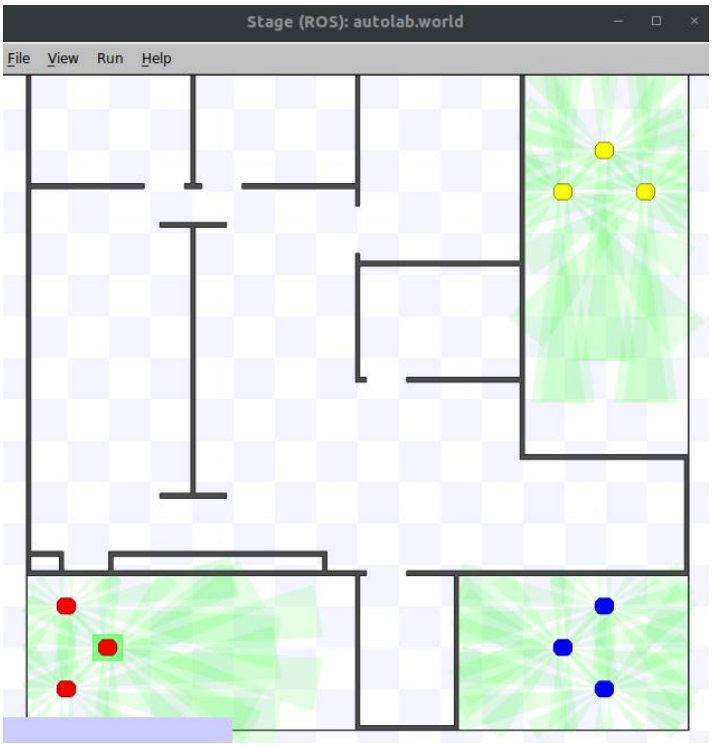


Image 6. Stage simulator

For more information on Stage: http://wiki.ros.org/stage

14.3 Gazebo

One of the open source robotic simulators tightly integrated with ROS is Gazebo (http://gazebosim.org). Gazebo is a dynamic robotic simulator with a wide variety of robot models and extensive sensors support The functionalities of Gazebo can be added via plugins. The sensor values can be accessed to ROS through topics, parameters and services.

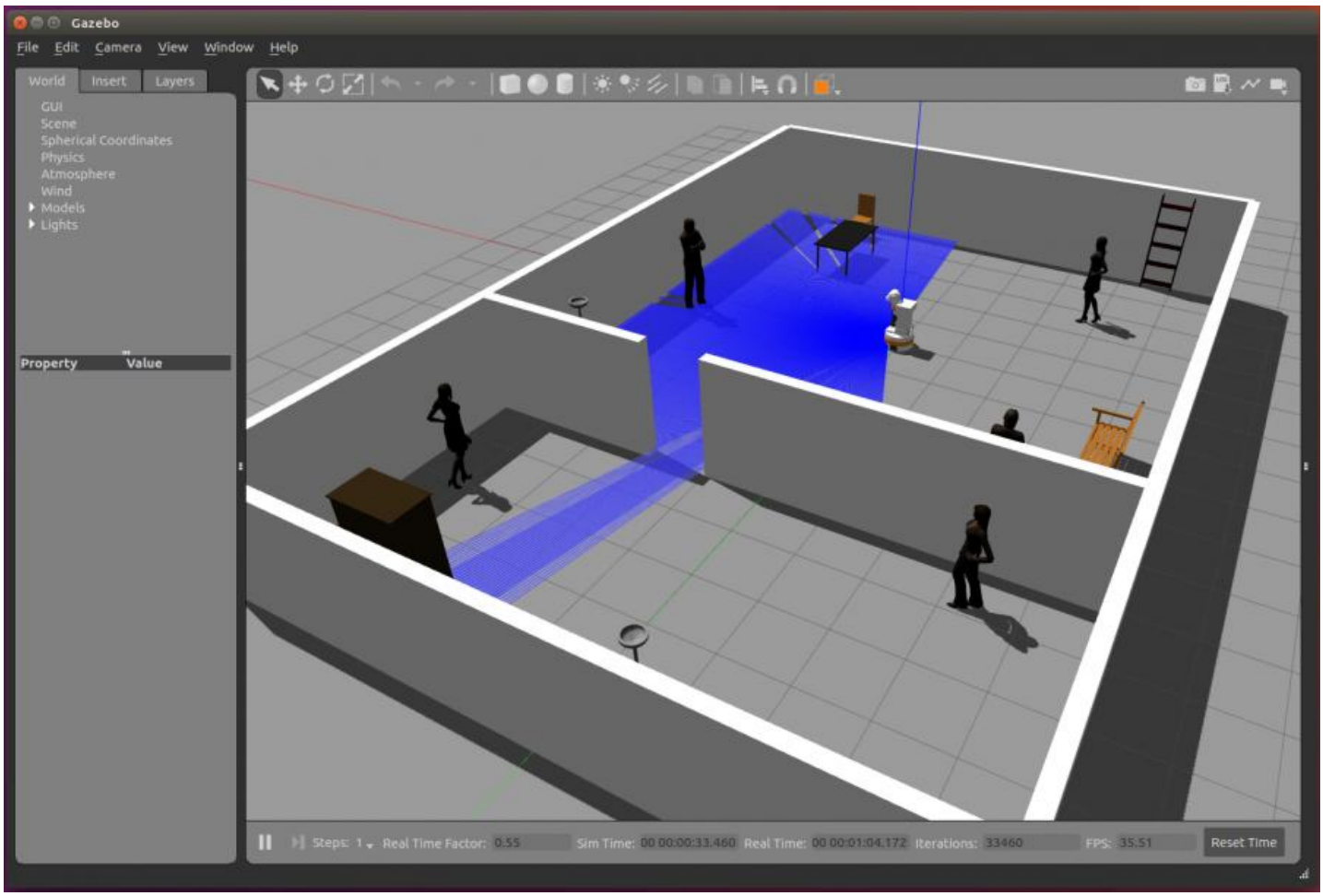


Image 7. Gazebo Simulator. Tiago Robot (PAL Robotics) with lidar (blue planar beam) activated

For installation: http://gazebosim.org/tutorials/?tut=ros_wrapper_versions

Considering that Gazebo simulator is installed, we could execute the following for turtlebot:

roslaunch turtlebot_gazebo turtlebot_world.launch

We can specify which world we want to use with argument world_file:

roslaunch turtlebot_gazebo turtlebot_world.launch world_file:=<full path to the world file>

You can find existing world files in this folder: */opt/ros/indigo/share/turtlebot_gazebo/worlds*

Launch an empty world:

roslaunch turtlebot_gazebo turtlebot_world.launch world_file:=/opt/ros/kinetic/share/turtlebot_gazebo/worlds/empty.world

Get a list of model names used in the simulation:

rostopic echo /gazebo/model_states -n1

To delete an object from the scenario:

rosservice call /gazebo/delete_model "model_name: '<model name>'"

14.3.1 Creating/ modifying worlds

Here you could find information on:

a) Editing worlds: http://learn.turtlebot.com/2015/02/03/6/
b) Creating Worlds: http://gazebosim.org/tutorials?tut=build_world&cat=build_world
c) Model editor: http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b3

After creating the world and saving it, it is needed to open the file and delete the robot object, as it will be repeated and cause problems when opening the world.

```
<model name='mobile_base'>
../..
```

```
</model>
```

If we store the worlds for example at ~/gazebo_worlds, a calling to gazebo with a new world may look like:

user/gazebo_worlds/Aisle_bs.world

14.3.2 Adding Laser Finder to Turtlebot in Gazebo Simulation

Here are the instructions to follow[19] (*based on:* https://bharat-robotics.github.io/blog/adding-hokuyo-laser-to-turtlebot-in-gazebo-for-simulation/)

It is better to create a new package to test the code rather than changing the turtlebot package.

cd ~/catkin_ws/src

Clone the turtlebot source into this folder:

git clone https://github.com/turtlebot/turtlebot.git

To use that new enviroment, you will need to source the new setup.bash from it. Source that with the following:

source ~/catkin_ws/devel/setup.bash

Actually, we can add it to ~/.bashrc so that we do not have to source it everytime.

Installing Hokuyo Related Packages

We must install hokuyo_node package as

sudo apt-get install ros-kinetic-urg-node (kinetic)
sudo apt-get install ros-kinetic-hokuyo3d

or

sudo apt-get install ros-indigo-hokuyo-node (indigo)

Creating the Hokuyo Description

We will be using mesh file *hokuyo.dae* for a visual representation of Hokuyo Laser Range Finder.

First we should add the *hokuyo.dae* file into ~/catkin_ws/src/turtlebot/turtlebot_description/meshes/sensors from

/opt/ros/**indigo**/share/gazebo_plugins/test/multi_robot_scenario/meshes/laser (if indigo. Other version of gazebo)

or

/opt/ros/**kinetic**/share/gazebo_plugins/test/multi_robot_scenario/meshes/laser (if kinetic. Other version of gazebo)

We'll start by creating a description of the Hokuyo physical specifications like size and position on our Turtlebot. For that, create a new file

~/catkin_ws/src/turtlebot/turtlebot_description/urdf/sensors/hokuyo.urdf.xacro

and add following to it.

```
<?xml version="1.0"?>
<robot name="sensor_hokuyo" xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:include filename="$(find turtlebot_description)/urdf/turtlebot_gazebo.urdf.xacro"/>
  <xacro:include filename="$(find turtlebot_description)/urdf/turtlebot_properties.urdf.xacro"/>

  <xacro:macro name="sensor_hokuyo" params="parent">
    <link name="hokuyo_link">
      <collision>
```

```
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <box size="0.1 0.1 0.1"/>
      </geometry>
    </collision>
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <mesh filename="package://turtlebot_description/meshes/sensors/hokuyo.dae"/>
      </geometry>
    </visual>
    <inertial>
      <mass value="1e-5" />
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
    </inertial>
  </link>
  <joint name="hokuyo_joint" type="fixed">
    <!--<axis xyz="0 0 1" />-->
    <origin xyz="0.08 0 0.430" rpy="0 0 0"/>
    <parent link="${parent}"/>
    <child link="hokuyo_link"/>
  </joint>
  <!-- Hokuyo sensor for simulation -->
  <turtlebot_sim_laser_range_finder/>
 </xacro:macro>
</robot>
```

The first elements of this block are an extra link (hokuyo_link) and joint (hokuyo_joint) added to the URDF file that represents the hokuyo position and orientation relative to turtlebot. In this xacro description sensor_hokuyo, we have passed parameter parent which functions as parent_link for hokuyo links and joints. We create a macro named *turtlebot_sim_laser_range_finder* that specifies all of the necessary information for the laser joint and link just before the closing tag.

Setting Up the Gazebo Plugin

Next we must define the Gazebo plugin that gives us the laser range finder functionality and publishes the laser scans to a ROS message. The file includes data that define the physical functional characteristics of the sensor, like angle of range, as well as the topic. It's name is:

~/catkin_ws/src/turtlebot/turtlebot_description/urdf/turtlebot_gazebo.urdf.xacro

This macro will set up the plugin in Gazebo when expanded from the hokuyo.urdf.xacro file.

Add the following code to it**,** after the </xacro:macro> tag closing the Kinect/Asus macro.

```
<xacro:macro name="turtlebot_sim_laser_range_finder">
 <!-- ULM-30LX Hokuyo Laser Range Finder -->
 <gazebo reference="hokuyo_link">
  <sensor type="gpu_ray" name="head_hokuyo_sensor">
   <pose>0 0 0 0 0 0</pose>
   <visualize>true</visualize>
   <update_rate>40</update_rate>
   <ray>
    <scan>
     <horizontal>
      <samples>1080</samples>
      <resolution>0.25</resolution>
      <min_angle>-2.3561945</min_angle>
      <max_angle>2.3561945</max_angle>
     </horizontal>
    </scan>
    <range>
     <min>0.10</min>
     <max>30.0</max>
     <resolution>0.01</resolution>
    </range>
    <noise>
     <type>gaussian</type>
     <!-- Noise parameters based on published spec for Hokuyo laser
          achieving "+-30mm" accuracy at range < 10m.  A mean of 0.0m and
```

```
          stddev of 0.01m will put 99.7% of samples within 0.03m of the true
          reading. -->
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
    <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_gpu_laser.so">
      <topicName>laserscan</topicName>
      <frameName>hokuyo_link</frameName>
    </plugin>
  </sensor>
</gazebo>
</xacro:macro>
```

We now add the Hokuyo urdf to the Turtlebot xacro library file in
/turtlebot_description/urdf/turtlebot_library.urdf.xacro.
Add the following line to the end of the file before the </robot> tag.

```
<!--  Hokuyo Laser Sensor-->
<xacro:include filename="$(find turtlebot_description)/urdf/sensors/hokuyo.urdf.xacro"/>
```

Creating a Description File for the Hokuyo-equipped Bot

Now that we've got our macros defined, we want to create a robot description file that actually uses them to specify the robot as having the Hokuyo equipped.

Navigate to the */turtlebot_description/robots* folder. Create a copy of whichever file corresponds to your robot's configuration (ours was kobuki_hexagons_kinect.urdf.xacro) as < base >_< stack >_hokuyo.urdf.xacro. As an example that left us with a copy named kobuki_hexagons_hokuyo.urdf.xacro. Now edit that file to add the following line.

```
<!--  Hokuyo Laser Sensor-->
<xacro:include filename="$(find turtlebot_description)/urdf/sensors/hokuyo.urdf.xacro"/>
<sensor_hokuyo parent="base_link"/>
```

Now we can use this file to launch our Turtlebot with the Hokuyo equipped and working. To specify you want to use the Hokuyo urdf you just created, run the following in the terminal every time we reboot the computer:

source ~/catkin_ws/devel/setup.bash
export TURTLEBOT_BASE=kobuki
export TURTLEBOT_STACKS=hexagons
export TURTLEBOT_3D_SENSOR="hokuyo"

Now we can launch the simulator:

roslaunch turtlebot_gazebo turtlebot_world.launch

To view actual laserscan readings, we must open rviz and then subscribe to /laserscan topic:

rosrun rviz rviz

Take care that this lidar is publishing in /laserscan and not in /scan

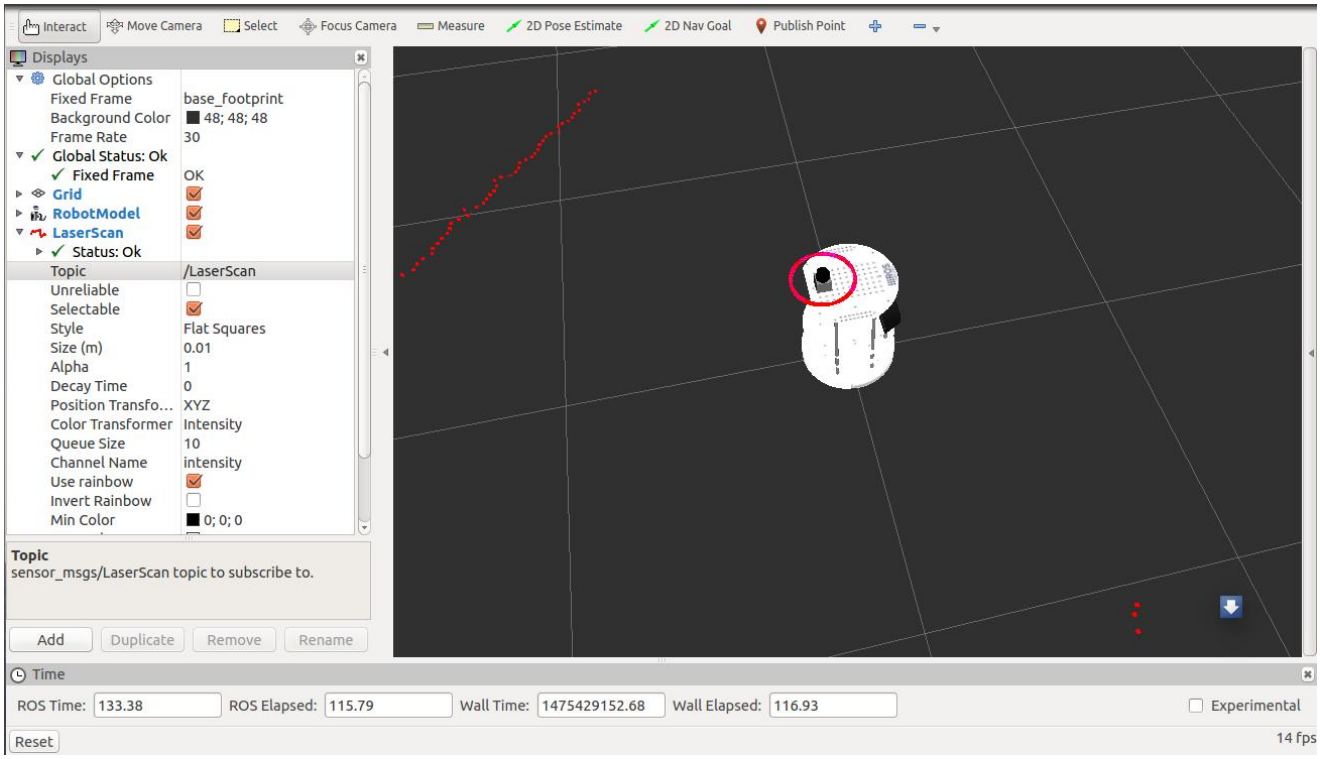


Image 8. Turtlebot Rviz Visualization of Hokuyo Range Finder Data

The stock files will have your kinect's virtual laser scan publishing on the same topic that other tutorials want to use, and I would rather the laser data would come from the hokuyo laser. This can be prevented by changing the topic that the Kinect laser publishes on with the following:

roscd to turtlebot_bringup/launch/
roscd turtlebot_bringup/launch/
edit 3dsensor.launch

Look for the line that looks like this:

```
<arg name="scan_topic" default="scan"/>
```

Remove it and replace it with:

```
<arg name="scan_topic" default="kinect_scan"/>
```

Save that file

Only if we are adding a real hokuyo to a real robot. No need if only used for simulations: Edit minimal.launch. We now need to add the hokuyo node to it. At the bottom of the file, right before </launch>, add the following:

```
<node name="laser_driver" pkg="hokuyo_node" type="hokuyo_node">  <param name="frame_id"
value="base_laser_link" /> </node>
```

The edit to minimal.launch should bring up the hokuyo node when you launch it.

copying files needed for gmapping:

cd ~/catkin_ws/src/turtlebot/turtlebot_bringup/launch/includes/3dsensor
sudo cp kinect.launch.xml hokuyo.launch.xml

cd /opt/ros/kinetic/share/turtlebot_navigation/launch/includes/amcl
sudo cp kinect_amcl.launch.xml hokuyo_amcl.launch.xml

cd /opt/ros/kinetic/share/turtlebot_navigation/param
sudo cp kinect_costmap_params.yaml hokuyo_costmap_params.yaml

cd /opt/ros/kinetic/share/turtlebot_navigation/launch/includes/gmapping
sudo cp kinect_gmapping.launch.xml hokuyo_gmapping.launch.xml

Main files explanation

- ~/catkin_ws/src/turtlebot/turtlebot_description/meshes/sensors/**hokuyo.dae** --- This is the mess file of the sensor. The graphic design.
- ~/catkin_ws/src/turtlebot/turtlebot_description/urdf/sensors/**hokuyo.urdf.xacro** --- description of the Hokuyo physical specifications like size and position on our Turtlebot. It is possible for example to rotate Pi/4 radians (45 degree) the laser sensor in yaw by changing the value of yaw (in radians):

```
<joint name="hokuyo_joint" type="fixed">
<!--<axis xyz="0 0 1" />-->
<origin xyz="0.08 0 0.430" rpy="0 0 ${M_PI/4}"/>
<parent link="${parent}"/>
<child link="hokuyo_link"/>
</joint>
```

  For example ${5*M_PI/4} is 225 degrees.

  Values xyz are the location of the sensor refereed to the origin of the robot coordinates. Values are in meters.

- ~/catkin_ws/src/turtlebot/turtlebot_description/urdf/**turtlebot_gazebo.urdf.xacro** --- Gazebo plugin that gives us the laser range finder and 3D camera functionality and publishes the laser scans and camera to a ROS message. The file includes data that define the physical functional characteristics of the sensor, like angle of range, as well as the topic.

Some other info here: https://duluthrobot.wordpress.com/2016/03/18/setup-files-for-hokuyo-lidar-on-turtlebot2/

15 Robot configuration (URDF)

Robot Configuration is extremely important in all the navigation modules. For instance, in the Mapping system, if you don't tell the system WHERE does your robot have the laser mounted at, which is it's orientation, the position of the wheels, etc., it won't be able to create a good and accurate Map.

Robot configuration and definition is done in the URDF files of the robot. URDF (Unified Robot Description Format) is an XML format that describes a robot model. It defines its different parts, dimensions, kinematics, dynamics, sensors, etc...

Several of these files detail the different parts of the robot, like the lidar, and other. These files are usually placed into a package named yourrobot_description (example: ~/catkin_ws/src/turtlebot/turtlebot_description)

More information here: http://wiki.ros.org/urdf/Tutorials

16 Coordinate Transforms (tf)

16.1 Overview

To correctly interpret a range scan produced by the laser, we need to know exactly where on the base the laser is attached. Is it mounted upside-down (which is not uncommon)? More generally, we could ask: what are the position and orientation of the laser with respect to the base?

A position-orientation pair is called a pose. For clarity, this kind of pose, which varies in six dimensions (three for translation plus three for rotation) is sometimes called a 6D pose. Given the pose of one thing relative to another, we can transform data between their frames of reference.

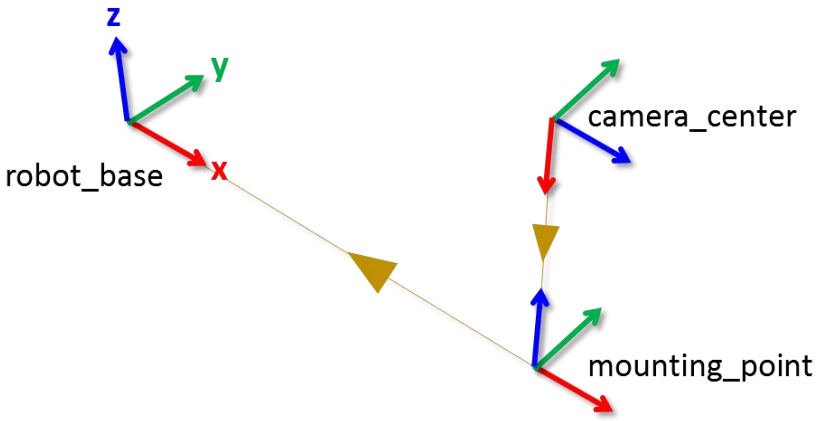


Image 9. Visual representation of three coordinates frames

There are many ways to manage coordinate frames and transforms between them. In ROS, continuing with the philosophy of keeping things small and modular, we take a distributed approach, using ROS topics to share transform data. Any node can be the authority that publishes the current information for some transform(s), and any node can subscribe to transform data, gathering from all the various authorities a complete picture of the robot. This system is implemented in the tf (short for transform) package, which is extremely widely used throughout ROS software.

We need names for coordinate frames. In tf, we use strings. The frame of the laser attached to the base might be called "laser" or, if there's the potential for confusion, "front_laser". You can pick any names you like.

We also need a message format to use when publishing information about transforms. In tf, we use tf/ tfMessage, sent over the /tf topic. Each tf/ tfMessage message contains a list of transforms, specifying for each one the names of the frames involved (referred to as parent and child), their relative position and orientation, and the time at which that transform was measured or computed.

tf also provides a set of libraries that can be used in any node to perform those common tasks. For example, if you create a tf listener in your node, then, behind the scenes, your node will subscribe to the /tf topic and maintain a buffer of all the tf/ tfMessage data published by other nodes in the system. Then you can ask questions of tf, like: Where is the laser with respect to the base?

Note: See tf Tools under Debugging Tools.

Some more information:

i. Introduction to tf: http://wiki.ros.org/tf/Tutorials/Introduction%20to%20tf
ii. Tutorial: http://wiki.ros.org/tf/Tutorials
iii. More about tf: http://wiki.ros.org/tf?distro=indigo

16.2 Frames of reference

The data captured by the different robot sensors must be referenced to a common frame of reference (usually the base_link) in order to be able to compare data coming from different sensors. The robot should publish the relationship between the main robot coordinate frame and the different sensors' frames using ROS tf.
Since the robot needs to be able to access this information anytime, we will publish this information to a transform tree. The transform tree is like a database where we can find information about all the transformations between the different frames (elements) of the robot.

Common frames of reference are[20]:

√ **base_link**: The coordinate frame called base_link is rigidly attached to the mobile robot base. The base_link can be attached to the base in any arbitrary position or orientation; for every hardware platform there will be a different place on the base that provides an obvious point of reference.

√ **odom**: The coordinate frame called odom is a world-fixed frame. The pose of a mobile platform in the odom frame can drift over time, without any bounds. This drift makes the odom frame useless as a long-term global reference. However, the pose of a robot in the odom frame is guaranteed to be continuous, meaning that the pose of a mobile platform in the odom frame always evolves in a smooth way, without discrete jumps. In a typical setup the odom frame is computed based on an odometry source, such as wheel odometry, visual odometry or an inertial measurement unit. The odom frame is useful as an accurate, short-term local reference, but drift makes it a poor frame for long-term reference.

√ **map**: The coordinate frame called map is a world fixed frame, with its Z-axis pointing upwards. The pose of a mobile platform, relative to the map frame, should not significantly drift over time. The map frame is not continuous, meaning the pose of a mobile platform in the map frame can change in discrete jumps at any time.

√ **earth**: This frame is designed to allow the interaction of multiple robots in different map frames. If the application only needs one map the earth coordinate frame is not expected to be present. In the case of running with multiple maps simultaneously the map and odom and base_link frames will need to be customized for each robot. If running multiple robots and bridging data between them, the transform frame_ids can remain standard on each robot if the other robots' frame_ids are rewritten. If the map frame is globally referenced the publisher from earth to map can be a static transform publisher. Otherwise the earth to map transform will usually need to be computed by taking the estimate of the current global position and subtracting the current estimated pose in the map to get the estimated pose of the origin of the map. In case the map frame's absolute positon is unknown at the time of startup, it can remain detached until such time that the global position estimation can be adaquately evaluated. This will operate in the same way that a robot can operate in the odom frame before localization in the map frame is initialized.

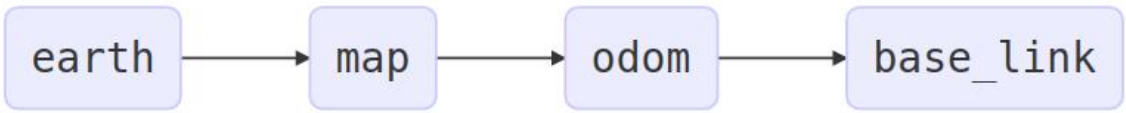


Image 10. Relationship between Frames

16.3 Adding frames of reference

Let's imagine you just mounted the laser on your robot, so the transform between your laser and the base of the robot is not set. What could you do? There are basically 2 ways of publishing a transform:

√ Use a static_transform_publisher
√ Use a transform broadcaster

The static_transform_publisher is a ready-to-use node that allows us to directly publish a transform by simply using the command line. The structure of the command is the next one:

static_transform_publisher x y z yaw pitch roll frame_id child_frame_id period_in_ms

Where:

- x, y, z are the offsets in meters
- yaw, pitch, roll are the rotation in radians
- period_in_ms specifies how often to send the transform

You can also create a launch file that launches the command above, specifying the different values in the following way:

```
<launch>
<node pkg="tf" type="static_transform_publisher" name="name_of_node"
args="x y z yaw pitch roll frame_id child_frame_id period_in_ms">
</node>
</launch>
```

Real example of Lidar in turtlebot. Added in the launch file of the Lidar node:

```
<node pkg="tf" type="static_transform_publisher" name="base_to_laser_broadcaster" args="0.211 0 0.04 3.926 0 0 base_link laser 100" />
```

It is in fact good idea to include this in the launch file of the new sensors.

17 ROS Navigation Stack

17.1 Overview

The Navigation Stack is a set of ROS nodes and algorithms which are used to autonomously move a robot from one point to another.

The Navigation Stack will take as input the current location of the robot, the desired location the robot wants to go (goal pose), the Odometry data of the Robot (wheel encoders, IMU, GPS...) and data from a sensor such as a Laser. In exchange, it will output the necessary velocity commands and send them to the mobile base in order to move the robot to the specified position.

As a pre-requisite for navigation stack use, the robot must be running ROS, have a tf transform tree in place, and publish sensor data using the correct ROS Message types. Also, the Navigation Stack needs to be configured for the shape and dynamics of a robot to perform at a high level[21].

Following there is a figure with the basic building blocks of the Navigational stack taken from ROS website (http://wiki.ros.org/navigation/Tutorials/RobotSetup).

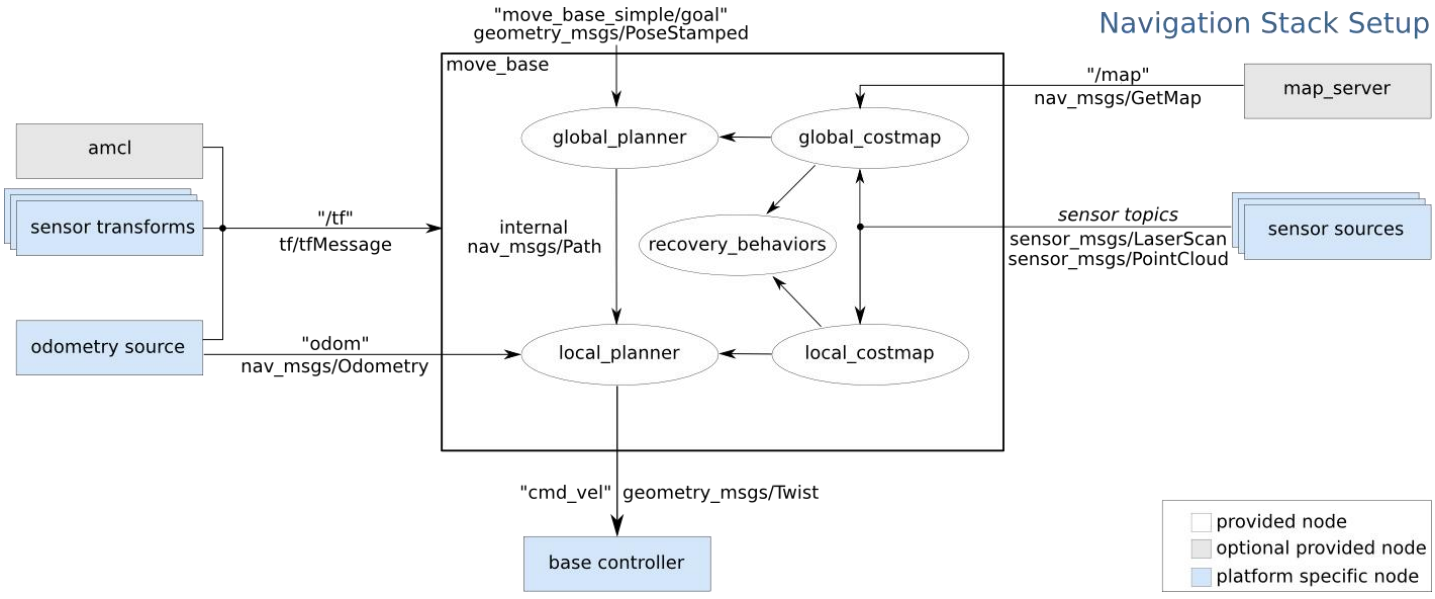


Image 11. basic building blocks of the Navigational stack

17.2 Odometry

Odometry is the use of data from motion sensors to estimate change in position over time. It is used in robotics by some legged or wheeled robots to estimate their position relative to a starting location. This method is sensitive to errors due to the integration of velocity measurements over time to give position estimates. Rapid and accurate data collection, instrument calibration, and processing are required in most cases for odometry to be used effectively[22].

The navigation stack uses tf to determine the robot's location in the world and relate sensor data to a static map. However, tf does not provide any information about the velocity of the robot. Because of this, the navigation stack requires that any odometry source publish both a transform and a nav_msgs/Odometry message over ROS that contains velocity information[23].

The robot's internal odometry can be supplemented with external measures of the robot's position and/or orientation. For example, one can use wall-mounted visual markers such as fiducials together with the ROS packages to provide a fairly accurate localization of the robot within a room. Some of the most common are:

i. ar_pose http://wiki.ros.org/ar_pose
ii. ar_kinect http://wiki.ros.org/ar_kinect
iii. ar_track_alvar http://wiki.ros.org/ar_track_alvar

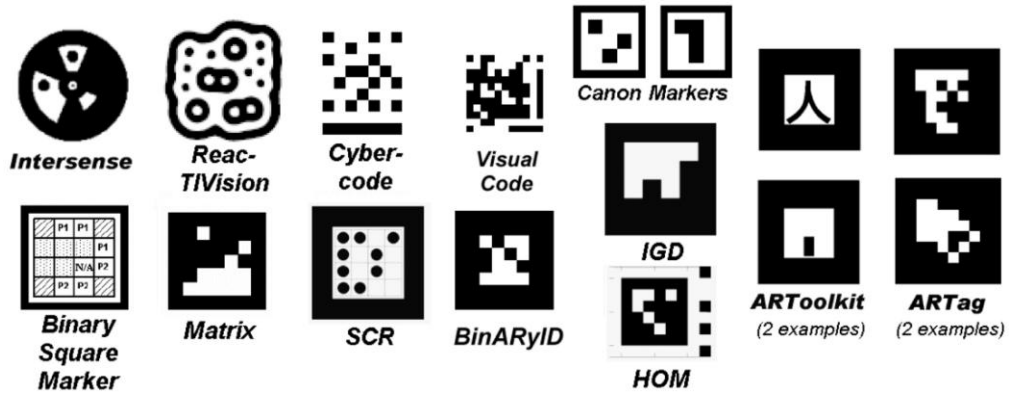


Image 12. Different types of fiducials

A similar technique uses visual feature matching without the need for artificial markers:

i. ccny_rgbd_tools http://wiki.ros.org/ccny_rgbd_tools
ii. rgbdslam http://wiki.ros.org/rgbdslam
iii. RTABMap http://introlab.github.io/rtabmap/

and yet another package uses laser scan matching:

i. laser_scan_matcher http://wiki.ros.org/laser_scan_matcher

Outdoor robots often use GPS to estimate position in addition to other forms of odometry.

ROS provides a message type to store the information; namely nav_msgs/Odometry. Its definition is as follows:

std_msgs/Header header uint32 seq time stamp string frame_id	It also provides a timestamp for each message so we know not only where we are but when.
string child_frame_id SLAM Background	*child_frame_id* define the reference frames we are using to measure distances and angles.
geometry_msgs/PoseWithCovariance pose geometry_msgs/Pose pose geometry_msgs/Point position float64 x float64 y float64 z geometry_msgs/Quaternion orientation float64 x float64 y float64 z float64 w float64[36] covariance	*PoseWithCovariance* sub-message records the position and orientation of the robot. Can be supplemented with a *covariance* matrix which measures the uncertainty in the various measurements.
geometry_msgs/TwistWithCovariance twist geometry_msgs/Twist twist geometry_msgs/Vector3 linear float64 x float64 y float64 z geometry_msgs/Vector3 angular float64 x float64 y float64 z	*TwistWithCovariance* component gives us the linear and angular speeds. Can be supplemented with a *covariance* matrix which measures the uncertainty in the various measurements.

float64[36] covariance	

By convention, odometry measurements in ROS use **/odom** as the parent frame id and **/base_link** (or **/base_footprint**) as the child frame id. While the **/base_link** frame corresponds to a real physical part of the robot, the **/odom** frame is defined by the translations and rotations encapsulated in the odometry data. These transformations move the robot relative to the **/odom** frame. If we display the robot model in RViz and set the fixed frame to the **/odom** frame, the robot's position will reflect where the robot "thinks" it is relative to its starting position.

17.3 SLAM Background

SLAM refers to Simultaneous Localization and Mapping. It is the process of building a map using range sensors (e.g. laser sensors, 3D sensors, ultrasonic sensors) while the robot is moving around and exploring an unknown area. The range sensor is used to detect the distance to obstacle whose estimated locations will be stored into a data structure (e.g/ 2D array) and when the robot is moving, it keeps updating this data structure my setting cell either occupied or empty based on the estimation of its location and the estimation of the distance to the obstacle. Usually, this process uses filtering techniques like Kalman filters or particle filters to improve the estimation of obstacle while moving and removing noise and errors from measurements. An example of error of measurement is the measure of odometry which known to be imprecise and is subject to cumulative errors over time. In addition, range sensors are another source of errors. The filtering technique allows to attenuate the effect of these errors onto the precision of the map. Large errors of odometry and/or range sensors will result into inaccurate and skewed maps.

For a more technical introduction to SLAM, consider the following references:

I. SLAM for Dummies[24]: An easy introduction to SLAM problem and solution. https://goo.gl/SrXuB3
II. Simultaneous Localisation and Mapping (SLAM)[25]: Part I The Essential Algorithms: a technical and mathematical introduction to SLAM. https://goo.gl/MTB6Uw

17.4 Mapping

In order to navigate consistently with a Robot, you need to have a map. This section describe the main packages and methods to get a proper map to support the navigation.

17.4.1 Gmapping package

This package contains a ROS wrapper for OpenSlam's Gmapping[26].

The gmapping package contains a ROS Node called *slam_gmapping*, which allows you to create a 2D map using the laser (/scan) and pose data (/tf) that your mobile robot is providing while moving around an environment. This node basically reads data from the laser and the transforms of the robot, and turns it into an occupancy grid map (OGM). In summary it will try to transform each incoming laser reading to the Odom frame.

The generated map is published during the whole process into the /map topic, which is the reason you could see the process of building the map with Rviz (because Rviz just visualizes topics).

The /map topic uses a message type of nav_msgs/OccupancyGrid, since it is an OGM. Occupancy is represented as an integer in the range {0, 100}. With 0 meaning completely free, 100 meaning completely occupied, and the special value of -1 for completely unknown.

This node is highly configurable and has lots of parameters you can change in order to improve the mapping performance. This parameters will be read from the ROS Parameter Server, and can be set either in the launch file itself or in a separated parameter files (YAML file). If you don't set some parameters, it will just take the default values.

Example of a YAML file with the parameters:

```
base_frame: base_footprint
odom_frame: odom
map_update_interval: 5.0
maxUrange: 6.0
maxRange: 8.0
minimumScore: 200
linearUpdate: 2.5
angularUpdate: 0.436
```

Example of a launch file that launch gmaping and load the YAML file parameters

```
<launch>
<arg name="scan_topic" default="/kobuki/laser/scan" />
<!-- Defining parameters for slam_gmapping node -->
<node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
output="screen">
<rosparam file="$(find my_mapping_launcher)/params/gmapping_params.yaml" command="load" />
<remap from="scan" to="$(arg scan_topic)"/>
</node>
</launch>
```

Other option:

```
<node pkg="gmapping" type="slam_gmapping" name="gmapping_thing" output="screen" >
<remap from="scan" to="youbot/scan_front" />
<param name="odom_frame" value="youbot/odom" />
<param name="base_frame" value="youbot/base_link" />
</node>
```

ROS wiki for more info: http://wiki.ros.org/gmapping

17.4.2 Building a map while navigating

Good tutorial: Make a map and navigate with it:
http://wiki.ros.org/turtlebot_gazebo/Tutorials/indigo/Make%20a%20map%20and%20navigate%20with%20it

Prepare the robot to be able to move around manually.

Turtlebot (Real)	**Turtlebot (Simulation)**
roscore	*source ~/catkin_ws/devel/setup.bash*

source ~/catkin_ws/devel/setup.bash *rosparam set /robot_state_publisher/use_tf_static false* *roslaunch turtlebot_bringup minimal.launch* *if Lidar*: *roslaunch rplidar_ros rplidar.launch* *if Kinect*: *export TURTLEBOT_3D_SENSOR=kinect* *roslaunch freenect_launch freenect.launch* *rosrun depthimage_to_laserscan* *depthimage_to_laserscan* *image:=/camera/depth/image_raw* *roslaunch turtlebot_teleop keyboard_teleop.launch or* *roslaunch turtlebot_teleop logitech.launch*	*rosparam set /robot_state_publisher/use_tf_static false* customize your simulated TurtleBot by setting TURTLEBOT_XXX environment variables *export TURTLEBOT_BASE=kobuki* *export TURTLEBOT_STACKS=hexagons* *export TURTLEBOT_3D_SENSOR="hokuyo"* *roslaunch turtlebot_gazebo turtlebot_world.launch* *roslaunch turtlebot_teleop keyboard_teleop.launch*

Start gmapping. As it subscribes to /scan while lidar is publishing in /laserscan in my simulation, y add a remapping:

rosrun gmapping slam_gmapping /scan:=/laserscan *(remapping not needed if Depth camera is used)*

Use RViz to visualize the map building process:

roslaunch turtlebot_rviz_launchers view_navigation.launch *(then I choose /laserscan as topic for LaserScan)*

Move the robot around to create the map.

roslaunch turtlebot_teleop keyboard_teleop.launch

Finally, save the map to disk:

rosrun map_server map_saver -f <your map name>

Check the map image:

eog <your map name>.pgm

More info[27]: http://faculty.rwu.edu/mstein/verbiage/EMARO%20Mapping%20Report.pdf

17.4.3 Building a map using rosbag

Collecting data

First, you need to start the robot drivers and then start the ROS node responsible for building the map. It has to be noted that to build a map ROS uses the gmapping software package, that is fully integrated with ROS. The gmapping package contains a ROS wrapper for OpenSlam's Gmapping. The package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called slam_gmapping. Using slam_gmapping, you can create a 2-D occupancy grid map (like a building floorplan) from laser and pose data collected by a mobile robot.

Prepare the robot to be able to move around manually. Considered a Turtlebot with an RPLidar Lidar unit (Not a hokuyo)

Turtlebot (Real)	**Turtlebot (Simulation)**
roscore *source ~/catkin_ws/devel/setup.bash* *rosparam set /robot_state_publisher/use_tf_static false* *roslaunch turtlebot_bringup minimal.launch* *roslaunch rplidar_ros rplidar.launch* *roslaunch turtlebot_teleop keyboard_teleop.launch or* *roslaunch turtlebot_teleop logitech.launch*	*source ~/catkin_ws/devel/setup.bash* *rosparam set /robot_state_publisher/use_tf_static false* customize your simulated TurtleBot by setting TURTLEBOT_XXX environment variables *export TURTLEBOT_BASE=kobuki* *export TURTLEBOT_STACKS=hexagons* *export TURTLEBOT_3D_SENSOR="hokuyo"* *roslaunch turtlebot_gazebo turtlebot_world.launch* *roslaunch turtlebot_teleop keyboard_teleop.launch*

Then load with rosbag

rosbag record -O data.bag /scan /tf /tf_static

Now, drive the robot around the world for a while. Try to cover as much of the map as possible, and make sure you visit the same locations a couple of times. Doing this will result in a better final map.
Once you've driven around for a while, use Ctrl-C to stop rosbag . Verify that you have a data bag called data.bag. You can find out what's in this bag by using the ros bag info command:

rosbag info data.bag

Once you have a bag that seems to have enough data in it, stop the simulator with a Ctrl-C in the terminal you ran roslaunch in.

Map generation

It's important to stop the simulator before starting the mapping process, because it will be publishing LaserScan messages that will conflict with those that are being replayed by rosbag. Now it's time to build a map. Start **roscore** in one of the terminals. In another terminal, we're going to tell ROS to use the timestamps recorded in the bag file, and start the slam_gmapping node:

rosparam set use_sim_time true
rosrun gmapping slam_gmapping

If your robot's laser scan topic is not called scan , you will need to tell slam_gmapping what it is by adding scan:=laser_scan_topic when you start the node. The mapper should now be running, waiting to see some data.

We're going to use rosbag play to replay the data that we recorded from the simulated robot:

rosbag play --clock data.bag

When it starts receiving data, slam_gmapping should start printing out diagnostic information. Sit back and wait until rosbag finishes replaying the data and slam_gmap ping has stopped printing diagnostics.

Note: if the slam_gmapping fails running (do not do anything), run the following command before the previous one:

rosbag play tf_static.bag --clock

At this point, your map has been built, but it hasn't been saved to disk. Tell slam_gmapping to do this by using the map_saver node from the map_server package. Without stopping slam_gmapping , run the map_saver node in another terminal:

rosrun map_server map_saver (use -f <filename> to generate the yaml and pgm with diferent name)

This will save two files to disk: map.pgm, which contains the map, and map.yaml, which contains the map metadata. Take a look, and make sure you can see these files. You can view the map file using any standard image viewer, such as eog:

eog map.pgm

17.4.4 Improving the mapping process

We can improve mapping quality by setting some of the gmapping parameters to different values, like the following:

rosparam set /slam_gmapping/angularUpdate 0.1
rosparam set /slam_gmapping/linearUpdate 0.1
rosparam set /slam_gmapping/lskip 10
rosparam set /slam_gmapping/xmax 10
rosparam set /slam_gmapping/xmin -10
rosparam set /slam_gmapping/ymax 10
rosparam set /slam_gmapping/ymin -10

These change how far the robot has to rotate (angularUpdate) and move (linearUpdate) before a new scan is considered for inclusion in the map, how many beams to skip when processing each LaserScan message (lskip), and the extent of the map (xmin , xmax , ymin , ymax).

Note that the parameter changes only affect slam_gmapping , so you could use them with the original data bag you collected, without driving the robot around again.

The ROS Navigation Stack is generic, that means, it can be used with almost any type of moving robot, but there are some hardware considerations that will help the whole system to perform better, so they must be considered. These are the requirements:

- The Navigation package will work better in differential drive and holonomic robots. Also, the mobile robot should be controlled by sending velocity commands in the form:
  x, y (linear velocity)
  z (angular velocity)
- The robot should mount a planar laser somewhere around the robot. It is used to build the map of the environment and perform localization.
- Its performance will be better for square and circular shaped mobile bases.

17.4.5 Starting a Map Server and Looking at a Map

Once you have a map, you need to make it available to ROS. You do this by running the map_server node from the map_server package, and pointing it at the YAML file for a map you have already made. As explained earlier, this YAML file contains the filename for the image that represents the map and additional information about it, like the resolution (meters per pixel), where the origin is, what the thresholds for occupied and open space are, and whether the image uses white for open space or occupied space.

With roscore running, you can start a map server like this:

rosrun map_server map_server map.yaml

Now, in another terminal, start up an instance of rviz :

rosrun rviz rviz

Add a display of type Map, and set the topic name to /map . Make sure that the fixed frame is also set to /map.

17.4.6 map_server package

map_server provides the map_server ROS Node, which offers map data as a ROS Service. It also provides the map_saver command-line utility, which allows dynamically generated maps to be saved to file[28].

map_saver command-line utility

Allows us to access the map data from a ROS Service, and save it into a file.

When you request the *map_saver* to save the current map, the map data (from /map topic) is saved into two files: one is the YAML file, which contains the map metadata and the image name, and second is the image itself (.PGM), which has the encoded data of the occupancy grid map.

YAML File

The YAML File generated will contain the 6 following fields:

```
image: my_map.pgm
resolution: 0.050000
origin: [-15.400000, -13.800000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Image 13. YAML file example

√ **image**: Name of the file containing the image of the generated Map.

√ **resolution**: Resolution of the map (in meters/pixel).

√ **origin**: Coordinates of the lower-left pixel in the map. This coordinates are given in 2D (x,y) in meters, considering the centre of the map being 0,0 (Therefore if the resolution is 0.05 and the image is 4000x2000 pixels, the coordinates would be x=-4000*0.05/2 and y=-2000*0.05/2. The third value indicates the rotation. If there's no rotation, the value will be 0.

√ **occupied_thresh**: Pixels which have a value greater than this value will be considered as a completely occupied zone.

√ **free_thresh**: Pixels which have a value smaller than this value will be considered as a completely free zone.

√ **negate**: Inverts the colours of the Map. By default, white means completely free and black means completely occupied.

Image File (PGM)

PGM (Portable Gray Map) file. A PGM is a file that represents a grayscale image. So, if you want to visualize this file properly, you should open it with an image editor.
The image describes the occupancy state of each cell of the world in the color of the corresponding pixel. Whiter pixels are free, blacker pixels are occupied, and pixels in between are unknown. Color and grayscale images are accepted, but most maps are gray (even though they may be stored as if in color). Thresholds in the YAML file are used to divide the three categories.

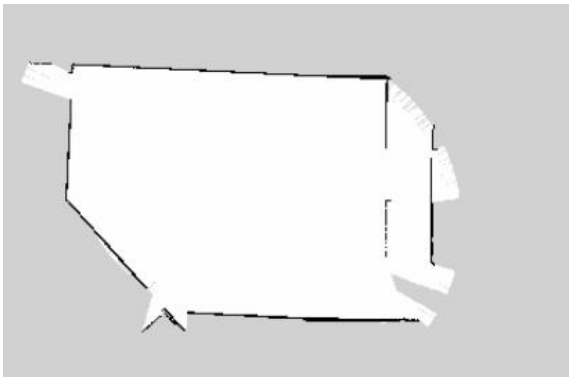


Image 14. Example of PGM map file

You can take the map generated and modify it manually, just by using an image editor. You can include forbidden areas, or delete people and other stuff included in the map.

map_server node

This node reads a map file from the disk and provides the map to any other node that requests it via a ROS Service. It publish in two topics: /map_metadata and /map

The current implementation of the map_server converts color values in the map image data into ternary occupancy values: free (0), occupied (100), and unknown (-1). Future versions of this tool may use the values between 0 and 100 to communicate finer gradations of occupancy.

Usage:

map_server <map.yaml>

Example:

rosrun map_server map_server mymap.yaml

Note that the map data may be retrieved via either latched topic (meaning that it is sent once to each new subscriber), or via service.

17.4.7 Editing the map (with Gimp)

pgm image map has a lot of noise data. The image could be improved, cropped and rotated adequately. For that we could use Gimp (or any other image editor)

Some edits to do on map:

Rotate
Usually the map is rotated. We could try to align it with the field of view of the monitor.
Zoom the area of the map
Select Layer, Transform, Arbitrary Rotation and move the slider to rotate it accordingly.

Crop
Crop the map to the area with relevant information.
Use the Rectangle selection to select the area to crop. Then menu Image, Crop to selection.

Clear areas
There are many noisy points in areas that we know there are no obstacles.
Use the color picker tool to take the color of a clear area.
Use pencil tool and select thickness of tip. Then paint the areas not clear that should be. Is possible to do lines with shift. This helps specially to make aisles clear.

Redefine edges

Some times edges are not clearly defined in areas that should be. Use pencil tool and black color for them.

Forbidden areas

Paint a line where the robot should not go.

Bend an Image

Sometimes due to odometry errors or lack of odometry calibration, long aisle may appear bended in the maps and may be corrected by Gimp.: https://docs.gimp.org/en/plug-in-curve-bend.html

Then Save the map with Gimp format, just in case we need to do modifications in the future, and export as pgm. Ensure the extension is .pgm and saves as raw, not ascii

Now we need to modify the yaml file. To know more about the Yaml file structure, read the section map_server package.

Edit the yaml file with any editor, like gedit. Modify the following data:

- √ **image**: Name of the file containing the image of the generated Map. Ensure the name of the file is the one exported from Gimp.
- √ **resolution**: Resolution of the map (in meters/pixel). Do not modify if you didn't change the size of the image. Just consider this value for the calculation of the next one.
- √ **origin**: Coordinates of the lower-left pixel in the map. This coordinates are given in 2D (x,y) in meters, If it is ok to have the origin of the coordinates of the map at the lower-left corner, just set as 0,0 if on the other hand you are considering the centre of the map being 0,0 (Therefore if the resolution is 0.05 and the image is 4000x2000 pixels, the coordinates would be x=-4000*0.05/2 and y=-2000*0.05/2. The third value indicates the rotation. If there's no rotation, the value will be 0. So the steps are to look for the properties of the file, size and calculate what is mentioned before.

17.4.8 Costmaps

The costmap is the data structure that represents places that are safe for the robot to be in a grid of cells. Usually, the values in the costmap are representing free space or places where the robot would be in collision[29].

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

a) The global navigation (global planner) is used to create paths for a goal in the map or a far-off distance.
b) The local navigation (local planner) is used to create paths in the nearby distances and avoid obstacles.

For this navigations two costmaps are used:

a) Global Costmap (*/move_base/global_costmap/costmap*) is based on the global map used. The global costmap is used for the global navigation.
b) Local Costmap (*/move_base/local_costmap/costmap*) is based on the actual inputs from sensors. The local costmap is used for the local navigation.

The costmap_2d package[30] uses sensor data and information from the static map to build a 2D or 3D occupancy grid of the data and inflate costs in a 2D costmap based on the occupancy grid and a user specified inflation radius.

Inflation is the process of propagating cost values out from occupied cells that decrease with distance. For this purpose, there are 5 specific symbols defined for costmap values:

1) "Lethal" cost means that there is an actual obstacle in a cell. So if the robot's center were in that cell, the robot would obviously be in collision.
2) "Inscribed" cost means that a cell is less than the robot's inscribed radius away from an obstacle. So the robot is certainly in collision with an obstacle if the robot center is in a cell that is at or above the inscribed cost.
3) "Possibly circumscribed“ cost is similar to inscribed, but using the robot's circumscribed radius as cutoff distance. Thus, if the robot center lies in a cell at or above this value, then it depends on the orientation of the robot whether it collides with an obstacle or not.
4) "Freespace" cost is assumed to be zero, and it means that there is nothing that should keep the robot from going there.

5) "Unknown" cost means there is no information about a given cell. The user of the costmap can interpret this as they see fit.

All other costs are assigned a value between "Freespace" and "Possibly circumscribed" depending on their distance from a "Lethal" cell and the decay function provided by the user.

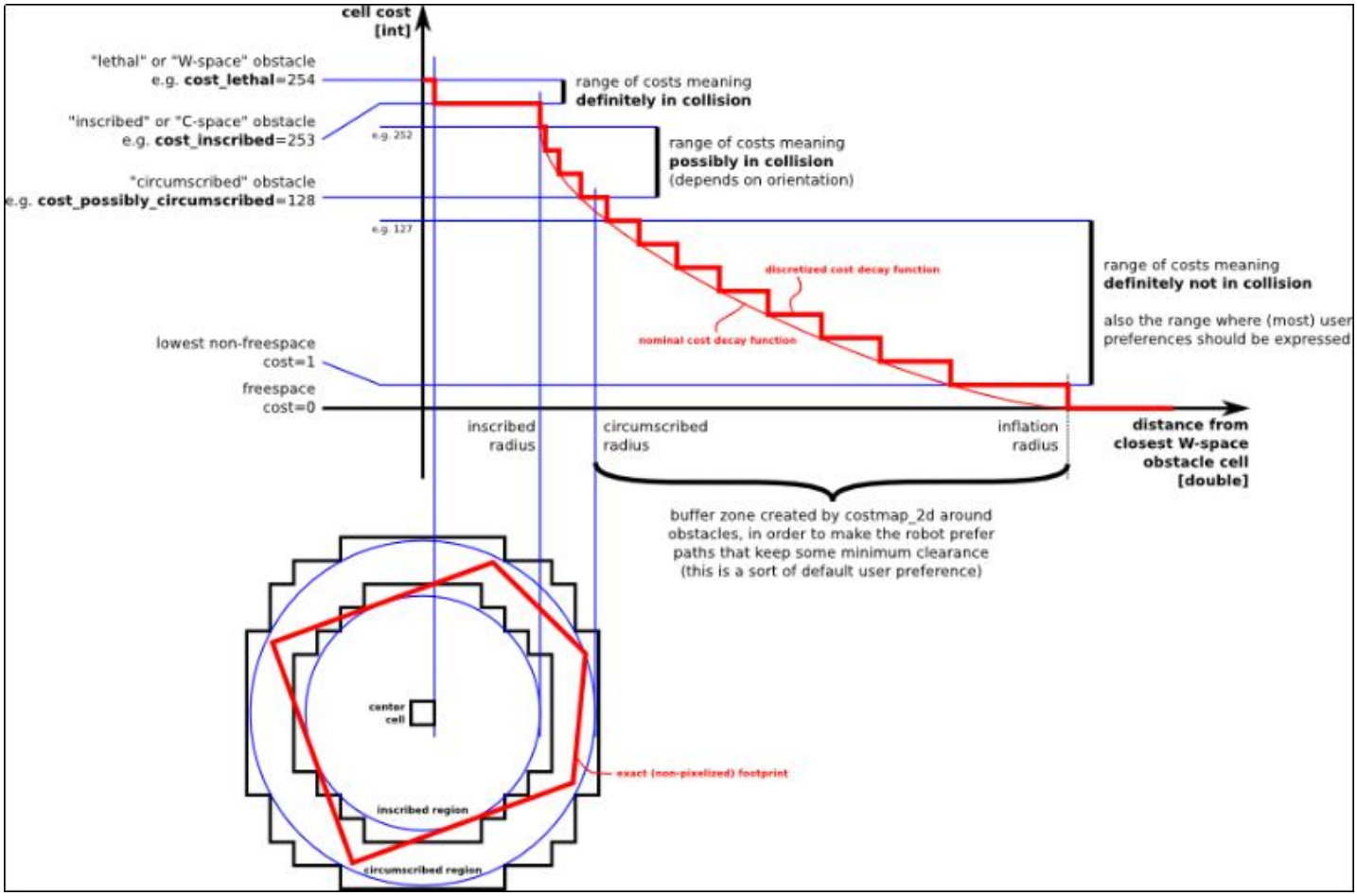


Image 15. Inflation. (C) Roi Yehoshua

Rviz allows to see both Costmaps. To see the costmap in rviz add a Map display.

i. To see the local costmap set the topic to: /move_base_node/local_costmap/costmap
ii. To see the global costmap set the topic to: /move_base_node/global_costmap/costmap

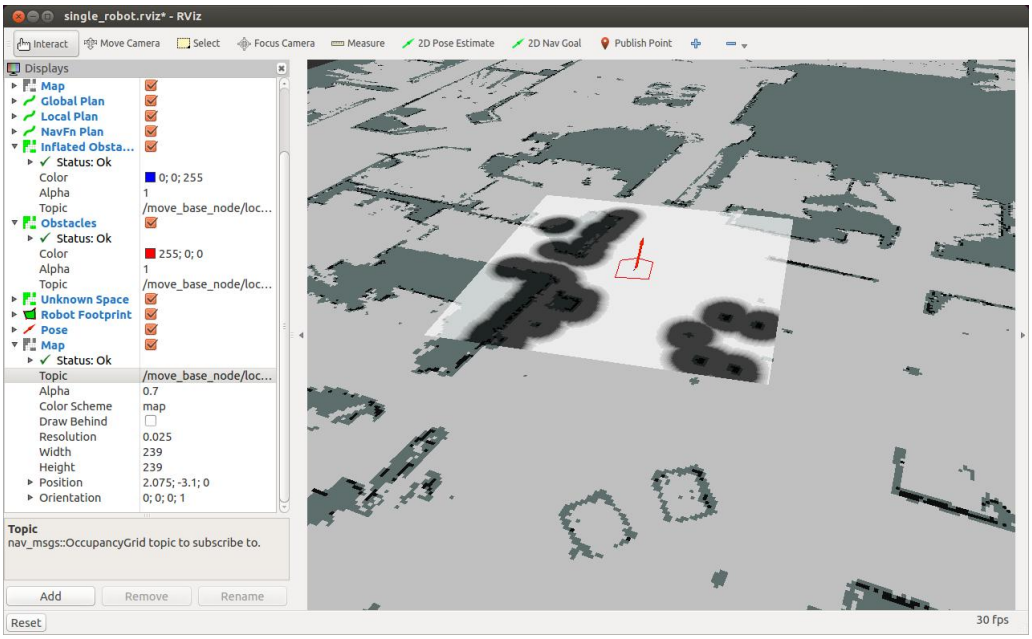


Image 16. Local Costmap

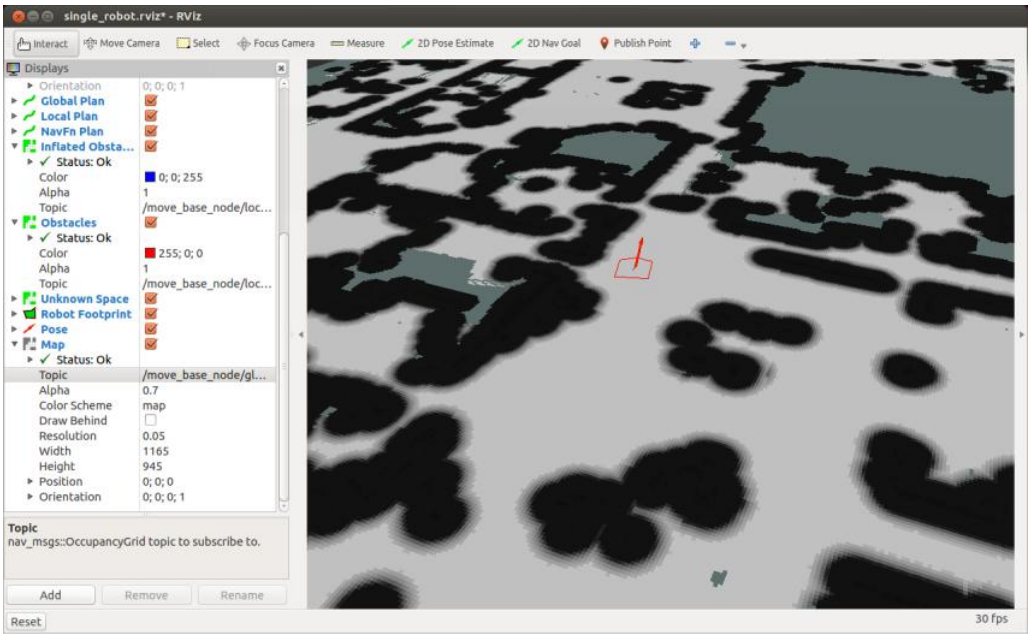


Image 17. Global Costmap

The costmap performs map update cycles at the rate specified by the *update_frequency* parameter.

17.5 Localization and Navigation

17.5.1 AMCL package

AMCL (Adaptive Monte Carlo Localization) is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox), which uses a particle filter to track the pose of a robot against a known map[31].

The AMCL package provides the **amcl** node. This node subscribes to the data of the laser, the laser-based map, and the transformations of the robot, and publishes its estimated position in the map. On startup, the amcl node initializes its particle filter according to the parameters provided.

We could set up an initial pose by using the 2D Pose Estimate tool in Rviz (which published that pose to the **/initialpose** topic).

The amcl node reads data published into the laser topic (**/scan**), the map topic (**/map**), and the transform topic (**/tf**), and published the estimated pose where the robot was in to the **/amcl_pose** and the **/particlecloud** topics.

If the particles covariance (published into the /amcl_pose topic) is less than 0.65, this means that the robot has localized itself correctly. The only values that you need to pay attention to is the first one (which is the covariance in x), the 8th one (which is the covariance in y), and the last one (which is the covariance in z)

This node is also highly customizable and we can configure many parameters in order to improve its performance. These parameters can be set either in the launch file itself or in a separate parameters file (YAML file). You can have a look at a complete list of all of the parameters that this node has here: http://wiki.ros.org/amcl

Example of a YAML file with the parameters:

```
use_map_topic: true
odom_model_type: diff
odom_frame_id: odom
```

```
gui_publish_rate: 10.0
min_particles: 500
max_particles: 2000
kld_err: 0.05
update_min_d: 0.25
update_min_a: 0.2
resample_interval: 1
transform_tolerance: 1.0

laser_max_beams: 60
laser_max_range: 12.0
laser_z_hit: 0.5
laser_z_short: 0.05
laser_z_max: 0.05
laser_z_rand: 0.5
```

Example of a launch file that launches *map_server* node, *amcl* node and load the YAML file parameters

```
<?xml version="1.0"?>
<launch>

<arg name="scan_topic" default="scan" />
<arg name="map_file" default="$(find husky_navigation)/maps/my_map.yaml"/>

<node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

<node pkg="amcl" type="amcl" name="amcl">
<rosparam file="$(find my_amcl_launcher)/config/my_amcl_params.yaml" command="load" />
<remap from="scan" to="$(arg scan_topic)"/>
</node>

</launch>
```

17.5.2 Localize the robot in the map

In this section, we'll see how we can use the ROS ***amcl*** package to localize the robot in a map.

Prepare the robot.

Turtlebot (Real)	**Turtlebot (Simulation)**
roscore *source ~/catkin_ws/devel/setup.bash* *rosparam set /robot_state_publisher/use_tf_static false* *roslaunch turtlebot_bringup minimal.launch* *roslaunch rplidar_ros rplidar.launch* *rosrun map_server map_server map.yaml (map server)* *roslaunch turtlebot_navigation amcl_demo.launch* *map_file:=/home/turtlebot<map YAML file>* *roslaunch turtlebot_rviz_launchers* *view_navigation.launch*	*source ~/catkin_ws/devel/setup.bash* *rosparam set /robot_state_publisher/use_tf_static false* customize your simulated TurtleBot by setting TURTLEBOT_XXX environment variables *export TURTLEBOT_BASE=kobuki* *export TURTLEBOT_STACKS=hexagons* *export TURTLEBOT_3D_SENSOR="hokuyo"* *rosrun map_server map_server map.yaml (map server)* *roslaunch turtlebot_gazebo amcl_demo.launch* *map_file:=<full path to your map YAML file>* *roslaunch turtlebot_rviz_launchers* *view_navigation.launch*

The costmap is available on a topic. In this case, the topic is /move_base/local_costmap/costmap. Should be configured on rviz in Global Map - Costmap - Topic. Also /move_base/local_costmap/costmap

Move the robot around

roslaunch turtlebot_teleop keyboard_teleop.launch

We will see the green arrows as the possible locations of the robot in the rviz.

You can help the robot localize by providing it its location on the map. To do that, go to the Rviz app. Then press the button *2D Pose Estimate* and go to the map, click on the actual location and drag to set orientation (arrow). The green arrows will be spread around that location.

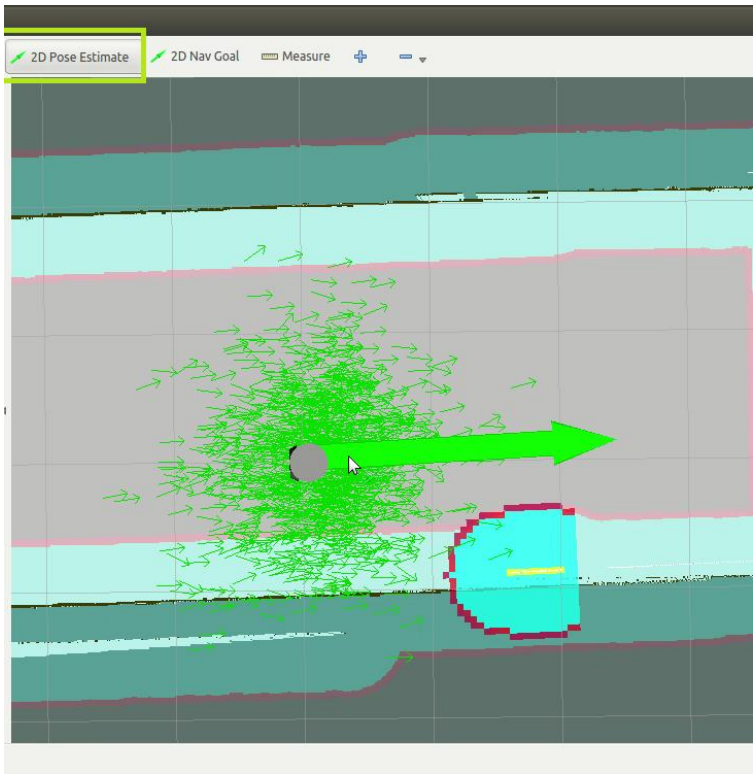


Image 18. Rviz 2D Pose Estimate

After this, move the robot around with the joystick, so the pose is better adjusted to the map (arrows converge).

17.5.3 Set a goal for the robot using Rviz

Considering we are in Rviz after following the steps described in previous section, after the AGV really knows were it is, we set a new destination clicking on *2D Nav Goal* on top of Rviz screen and then click on destination and drag to set final orientation. Then a color grid is shown as well as the path and the robot start moving towards the new goal avoiding obstacles.

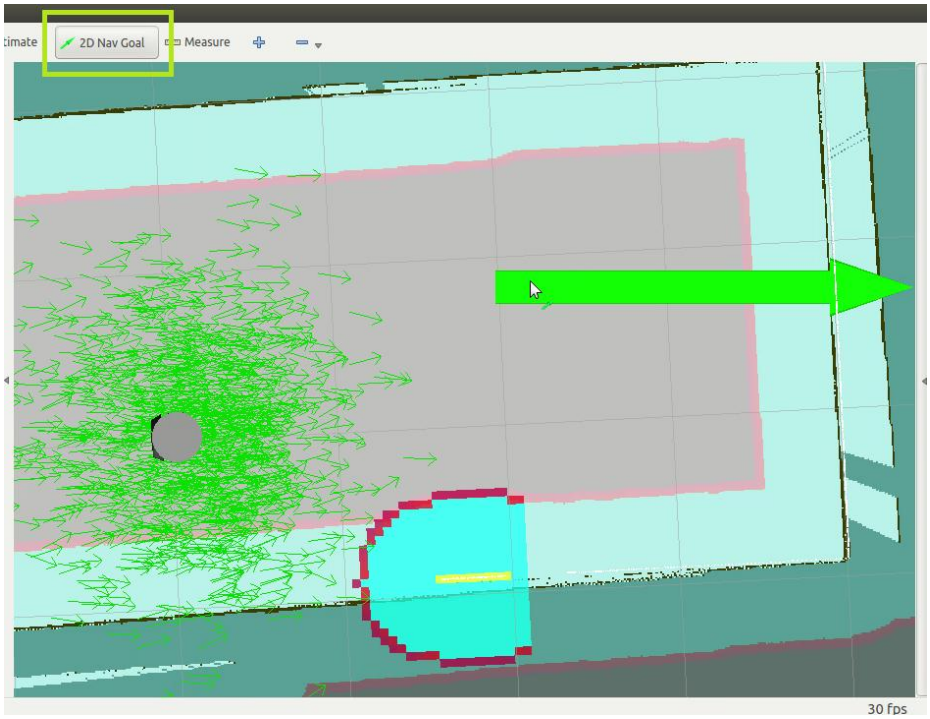


Image 19. Rviz 2D Nav Goal

Note: If the robot do not move automatically after setting destination is probably because we need to close the keyboard_teleop

The teleoperation can be run simultaneously with the navigation stack. It will override the autonomous behavior if commands are being sent. It is often a good idea to teleoperate the robot after seeding the localization to make sure it converges to a good estimate of the position ---> But it is better if you use the joystick

Note: Some times the navigation fails in simulation, making the robot spin a lot. Could be controlled by reducing the navigation speed with rqt_reconfigure

17.5.4 Set a goal for the robot with Python

This node, has a list of goal poses that it cycles through in order, calling the move_base action repeatedly and then waiting for it to terminate.

```
#!/usr/bin/env python
import rospy
import actionlib
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
waypoints = [   #A list of the waypoints for the robot to patrol
[(2.1, 2.2, 0.0), (0.0, 0.0, 0.0, 1.0)],
[(6.5, 4.43, 0.0), (0.0, 0.0, -0.984047240305, 0.177907360295)]
]

def goal_pose(pose):
    # A helper function to turn a waypoint into a MoveBaseGoal
    goal_pose = MoveBaseGoal()
    goal_pose.target_pose.header.frame_id = 'map'
    goal_pose.target_pose.pose.position.x = pose[0][0]
    goal_pose.target_pose.pose.position.y = pose[0][1]
    goal_pose.target_pose.pose.position.z = pose[0][2]
    goal_pose.target_pose.pose.orientation.x = pose[1][0]
    goal_pose.target_pose.pose.orientation.y = pose[1][1]
    goal_pose.target_pose.pose.orientation.z = pose[1][2]
    goal_pose.target_pose.pose.orientation.w = pose[1][3]
    return goal_pose

if __name__ == '__main__':
    rospy.init_node('patrol')
    #Create a simple action client, and wait for the server to be ready.
    client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    client.wait_for_server()
    while True:
        for pose in waypoints:   #Loop through the waypoints, sending each as an action goal
            goal = goal_pose(pose)
            client.send_goal(goal)
            client.wait_for_result()
```

This code just repeatedly sends action goals to the move_base action and waits for them to complete. The waypoints are specified by position and a quaternion that represents rotation. You can specify the frame that

these coordinates are in as part of the MoveBaseGoal argument. In our case, we're using the map frame. However, if you wanted to go to an object, and that object had its own coordinate frame that ROS knew about, you could just as easily use that.

17.5.5 move_base package

The move_base package provides an implementation of an action (see the actionlib package) that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner to accomplish its global navigation task. It supports any global planner adhering to the nav_core::BaseGlobalPlanner interface specified in the nav_core package and any local planner adhering to the nav_core::BaseLocalPlanner interface specified in the nav_core package. The move_base node also maintains two costmaps, one for the global planner, and one for a local planner (see the costmap_2d package) that are used to accomplish navigation tasks[32].

move_base Action Server provides the following 5 topics:

√ move_base/goal (move_base_msgs/MoveBaseActionGoal) This topic is then used to provide the goal pose.
√ move_base/cancel (actionlib_msgs/GoalID)
√ move_base/feedback (move_base_msgs/MoveBaseActionFeedback)
√ move_base/status (actionlib_msgs/GoalStatusArray)
√ move_base/result (move_base_msgs/MoveBaseActionResult)

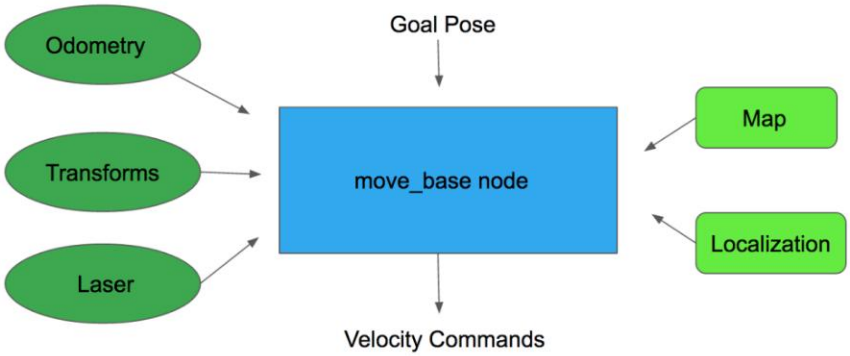
Keep in mind that in order to be able to send goals to the Action Server, the move_base node must be launched.

When this node receives a goal pose, it links to components such as the global planner, local planner, recovery behaviors, and costmaps, and generates an output, which is a velocity command with the message type geometry_msgs/Twist, and sends it to the /cmd_vel topic in order to move the robot.

See an example of code to move the robot to different waypoints in the Code Snippets section of this document.

The move_base wiki page (http://wiki.ros.org/move_base) lists all of the parameters you can set to tune the performance of the nav stack.

The main function of the move_base node is to move a robot from its current position to a goal position with the help of other Navigation nodes. This node links the global-planner and the local-planner for the path planning, connecting to the rotate-recovery package if the robot is stuck in some obstacle, and connecting global costmap and local costmap for getting the map of obstacles of the environment.



It is launched when executing:

roslaunch turtlebot_gazebo amcl_demo.launch map_file:=<full path to your map YAML file>

More info on http://wiki.ros.org/move_base

17.5.6 Global Planner

When a new goal is received by the move_base node, this goal is immediately sent to the planner. Then, the planner is in charge of calculating a safe path in order to arrive at that goal pose. This path is calculated before the robot starts moving, so it will **not** take into account the readings that the robot sensors are doing while moving.

There exist different planners. Depending on your setup (the robot you use, the environment it navigates, etc.), you would use one or another. Let's have a look at the most important ones.

Navfn Planner

navfn provides a fast interpolated navigation function that can be used to create plans for a mobile base. The planner assumes a circular robot and operates on a costmap to find a minimum cost plan from a start point to an end point in a grid. The navigation function is computed with Dijkstra's algorithm[33].

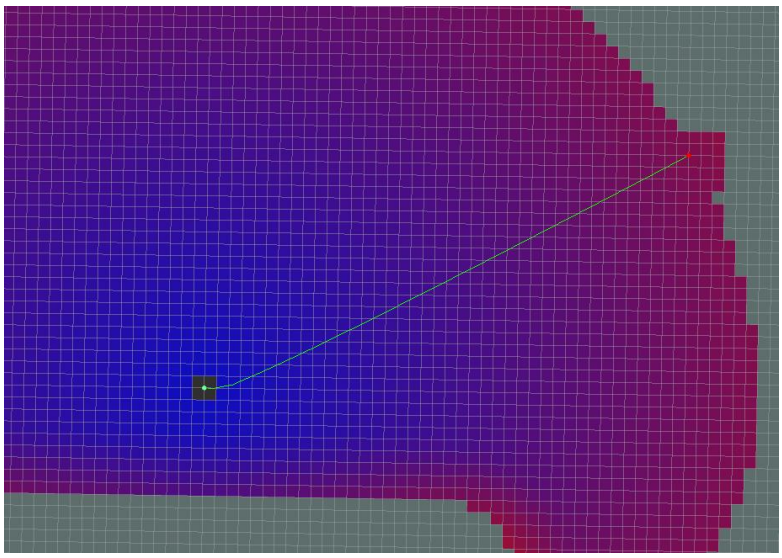


Image 20. Dijkstra's algorithm

For more information: http://wiki.ros.org/navfn

Carrot Planner

The carrot planner takes the goal pose and checks if this goal is in an obstacle. Then, if it is in an obstacle, it walks back along the vector between the goal and the robot until a goal point that is not in an obstacle is found. It, then, passes this goal point on as a plan to a local planner or controller. Therefore, this planner does not do any global path planning. It is helpful if you require your robot to move close to the given goal, even if the goal is unreachable. In complicated indoor environments, this planner is not very practical.

This algorithm can be useful if, for instance, you want your robot to move as close as possible to an obstacle (a table, for instance).

For more information[34]: http://wiki.ros.org/carrot_planner

Global Planner

This package provides an implementation of a fast, interpolated global planner for navigation. The global planner is a more flexible replacement for the navfn planner. It allows you to change the algorithm used by navfn (Dijkstra's algorithm) to calculate paths for other algorithms. These options include support for A∗, toggling quadratic approximation, and toggling grid path.

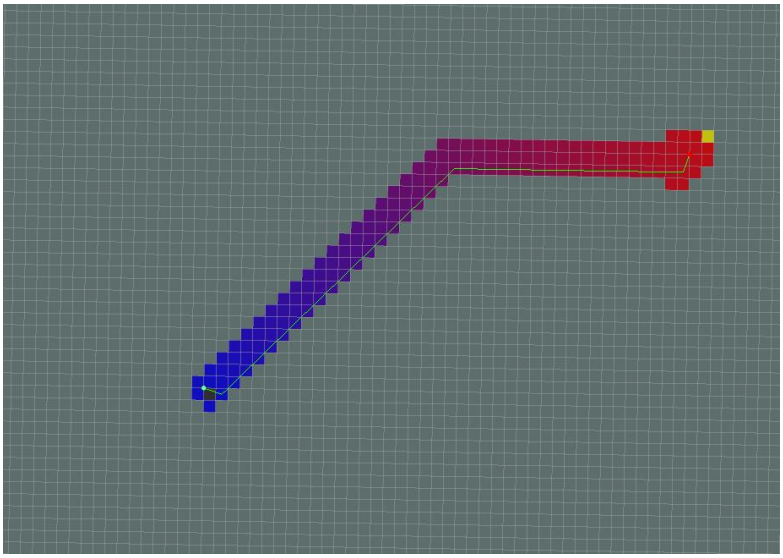


Image 21. A algorithm*

For more information: http://wiki.ros.org/global_planner

Change the Planner

The planner used by the move_base node is specified in the move_base parameters file. In order to do this, you will add one of the following lines to the parameters file:

√ base_global_planner: "navfn/NavfnROS" # Sets the Navfn Planner
√ base_global_planner: "carrot_planner/CarrotPlanner" # Sets the Carrot Planner
√ base_global_planner: "global_planner/GlobalPlanner" # Sets the Global Planner

To know the planner used:

rosparam get /move_base/base_global_planner

18 Time management

18.1 Delays control in a script

This is an example on how to control delays in a script without stopping the normal execution of the program. Basically what it does is to compare the actual timer with goal, that is the timer plus 1 second:

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

cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
rospy.init_node('red_light_green_light')

red_light_twist = Twist()
green_light_twist = Twist()
green_light_twist.linear.x = 0.5

driving_forward = True
light_change_time = rospy.Time.now()
rate = rospy.Rate(10)

while not rospy.is_shutdown():
if driving_forward:
cmd_vel_pub.publish(green_light_twist)
else:
cmd_vel_pub.publish(red_light_twist)
if light_change_time < rospy.Time.now():
driving_forward = not driving_forward
light_change_time = rospy.Time.now() + rospy.Duration(1) #Add one second to the goal
rate.sleep()
```

18.2 Time synchronization

Time synchronization between machines is often critical in a ROS network since frame transformations and many message types are timestamped. An easy way to keep your computers synchronized is to install the Ubuntu chrony package on both your desktop and your robot. This package will keep your computer clocks synchronized with Internet servers and thus with each other.

To install chrony, run the command:

```
$ sudo apt-get install chrony
```

After installation, the chrony daemon will automatically start and begin synchronizing your computer's clock with a number of Internet servers.

19 Debugging tools

ROS has a variety of GUI and command-line tools to inspect and debug messages. Let's look at some commonly used ones.

19.1 Rviz

Rviz (http://wiki.ros.org/rviz) is one of the 3D visualizers available in ROS to visualize 2D and 3D values from ROS topics and parameters. Rviz helps visualize data such as robot models, robot 3D transform data (TF), point cloud, laser and image data, and a variety of different sensor data.

It is launched by:

rosrun rviz rviz

Or by:

roslaunch turtlebot_rviz_launchers view_robot.launch

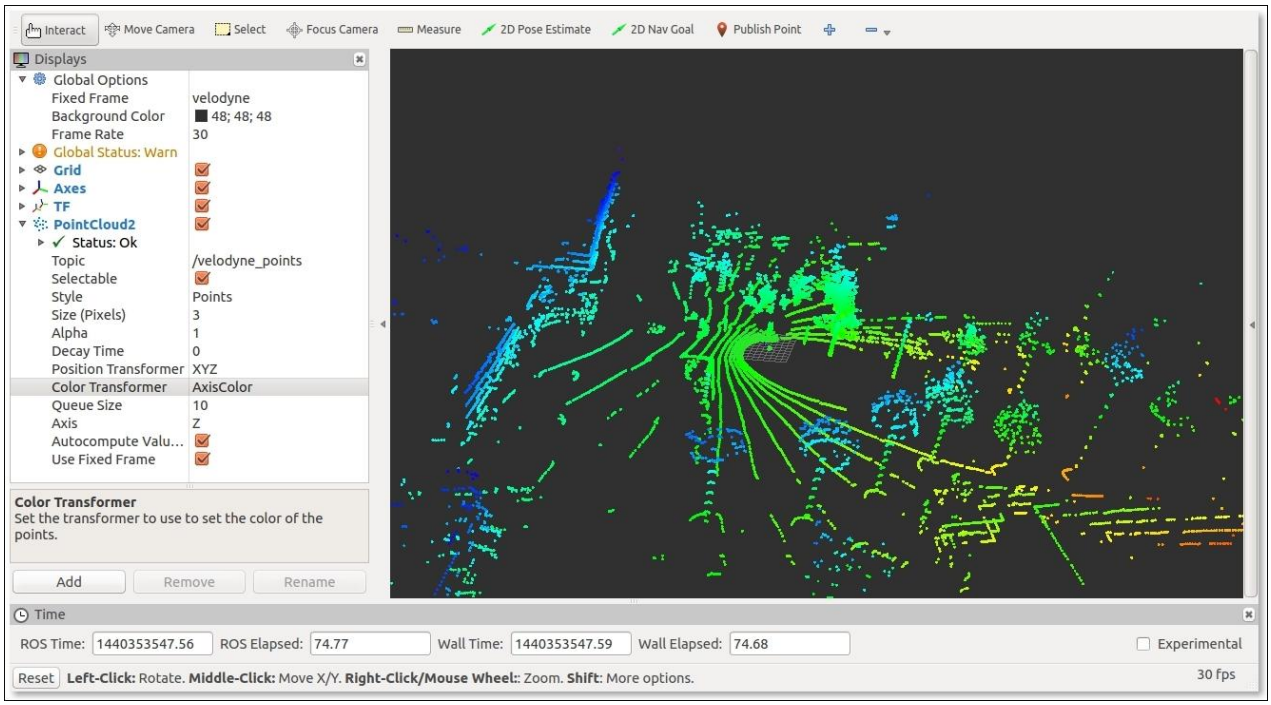


Image 22. Point cloud data visualized in Rviz

RVIZ is a tool that allows you to visualize Images, PointClouds, Lasers, Kinematic Transformations, RobotModels...The list is endless. You even can define your own markers.

More information here:

i. Tutorial: http://docs.ros.org/indigo/api/rviz/html/user_guide/
ii. RViz package: http://wiki.ros.org/rviz

19.2 rqt_plot

This is a very common need in any scientific discipline, but especially important in robotics. You need to know if your inclination is correct, your speed is the right one, the torque readings in an arm joint is above normal, or the laser is having anomalous readings. For all these types of data, you need a graphic tool that makes some sense of all the data you are receiving in a fast and real-time way. Here is where rqt_plot comes in handy. It could be called in a terminal as:

rqt_plot

In the new window that should pop up, a text box in the upper left corner gives you the ability to add any topic to the plot.

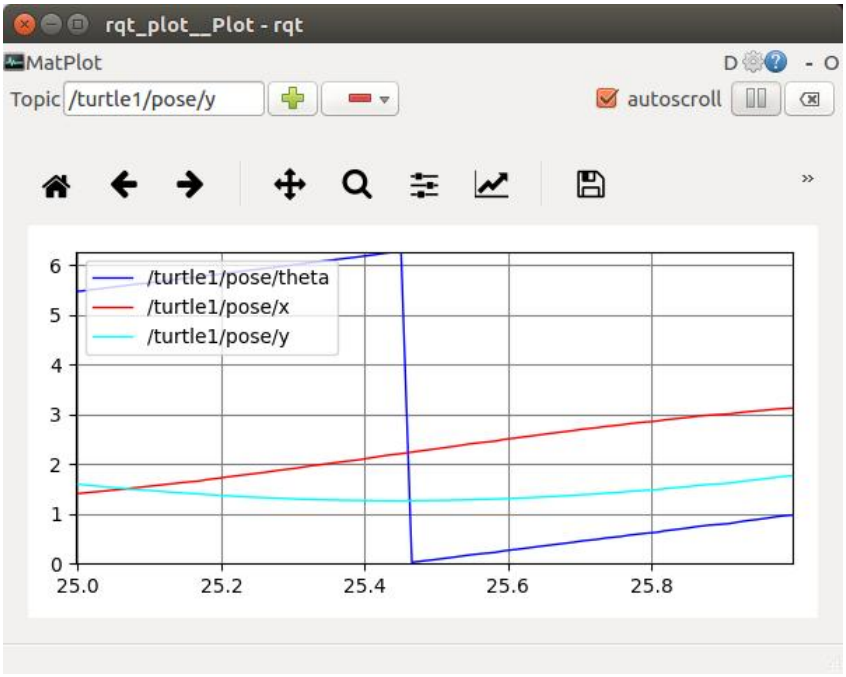


Image 23. rqt_plot

Example of call:

rqt_plot /laser_scan/ranges[100]

rqt_plot /cmd_vel_mux/input/navi/linear/x /cmd_vel_mux/input/navi/angular/z (linear/angular speed)

More information: http://wiki.ros.org/rqt_plot

19.3 rqt_graph

rqt_graph is a graph visualizer. It could be called in a terminal as:

rqt_graph

This will bring up a display that produces renderings of the connections between nodes. This renderings will not autorefresh, but you can click the refresh icon in the upper-left corner of the rqt_graph window when you add a

node to or remove one from the ROS graph by terminating (e.g., pressing Ctrl-C) or running (via rosrun) its program, and the graph will be redrawn to represent the current state of the system.

There are some filters that might be used.

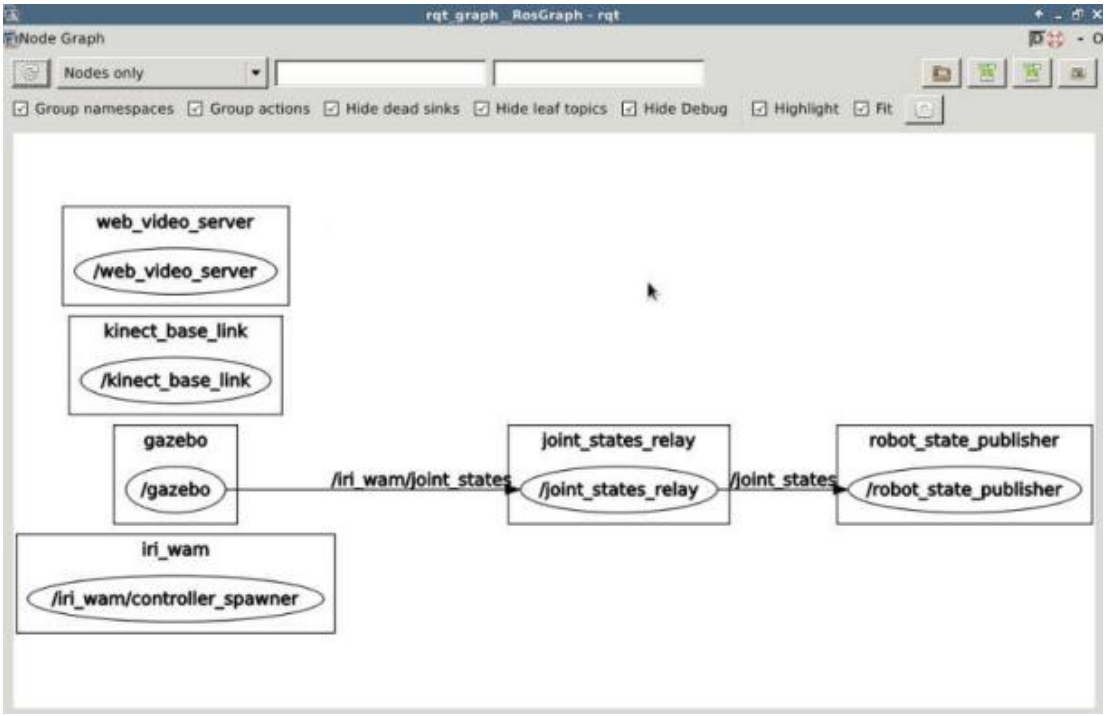


Image 24. rqt_graph

More information: http://wiki.ros.org/rqt_graph

19.4 rqt_reconfigure

ROS nodes store their configuration parameters on the ROS Parameter Server where they can be read and modified by other active nodes. You will often want to use ROS parameters in your own scripts so that you can set them in your launch files, override them on the command line, or modify them through rqt_reconfigure (formerly dynamic_reconfigure) if you add dynamic support.

ROS provides the command line tool *rosparam* for getting and setting parameters. However, parameter changes made this way will not be read by a node until the node is restarted.

The ROS rqt_reconfigure package (formerly called dynamic_reconfigure) provides an easy-to-use GUI interface to a subset of the parameters on the Parameter Server. It can be launched any time using the command:

$ rosrun rqt_reconfigure rqt_reconfigure

The rqt_reconfigure GUI allows you to change parameters for nodes dynamically— i.e., without having restart a node. However, there is one catch: only nodes that have been programmed using the rqt_reconfigure API will be visible in the rqt_reconfigure GUI. This includes most nodes in the key ROS stacks and packages such as Navigation, but many third-party nodes do not use the API and therefore can only be tweaked using the rosparam command line tool followed by a node restart.

NOTE: Unlike the dynamic_reconfigure package in previous ROS versions, rqt_reconfigure does not appear to dynamically detect new nodes if they are launched after you bring up the GUI. To see a freshly launched node in the rqt_reconfigure GUI, close the GUI then bring it back up again.

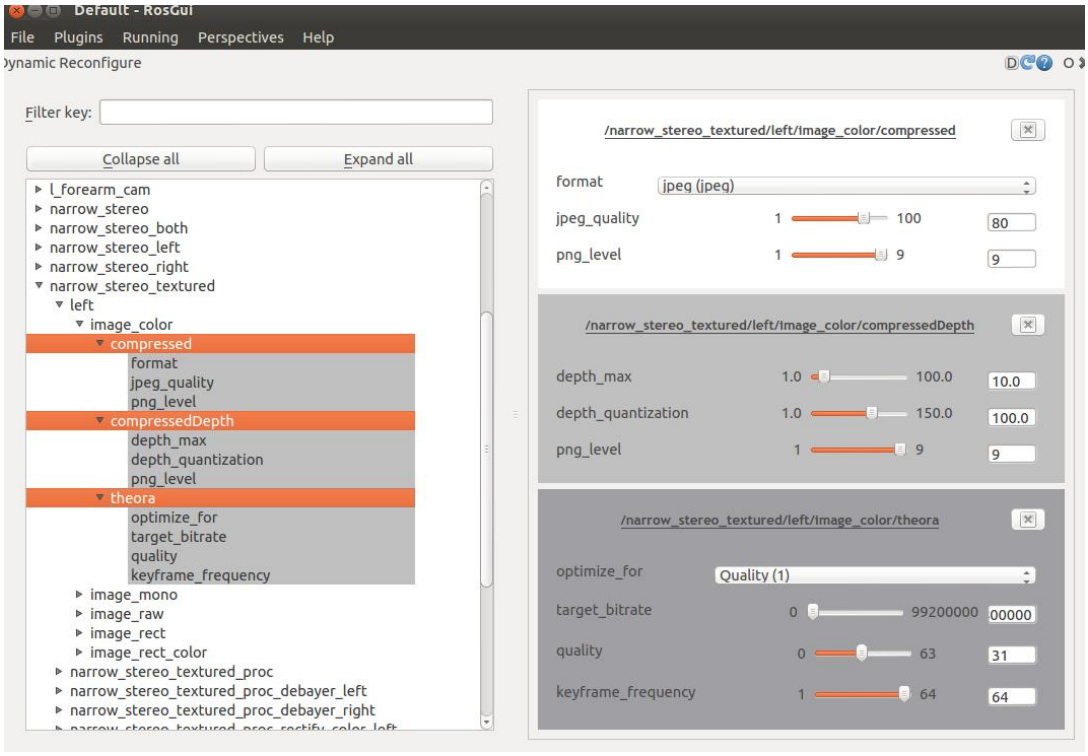


Image 25. rqt_reconfigure tool

Adding rqt_reconfigure support to your own nodes is not difficult and if you would like to learn how, refer to the step-by-step *Dynamic Reconfigure Tutorials* on the ROS Wiki.

More Information: http://wiki.ros.org/rqt_reconfigure

19.5 Including logs in python

ROS loging system use 5 levels of logging.

Use the Python module rospy to access the logging functionality in Python.

DEBUG ==> rospy.logdebug(msg, args)
INFO ==> rospy.loginfo(msg, args)
WARNING ==> rospy.logwarn(msg, args)
ERROR ==> rospy.logerr(msg, args)
*FATAL ==> rospy.logfatal(msg, *args)*

examples:

```
rospy.logdebug("There is a missing droid")
rospy.loginfo("The Emperors Capuchino is done")
rospy.logwarn("The Revels are coming time "+str(time.time()))
exhaust_number = random.randint(1,100)
port_number = random.randint(1,100)
rospy.logerr(" The thermal exhaust port %s, right below the main port %s", exhaust_number, port_number)
rospy.logfatal("The DeathStar Is EXPLODING")
rate.sleep()
rospy.logfatal("END")
```

with **rostopic echo /rosout** we could see all of the ROS logs in the current nodes, running in the system. Also rqt_console helps a lot on this matter.

19.6 rqt_console

rqt_console is a viewer in the rqt package that displays messages being published to rosout. It collects messages over time, and lets you view them in more detail, as well as allowing you to filter messages by various means. Help to see in a graphical way error log messages.

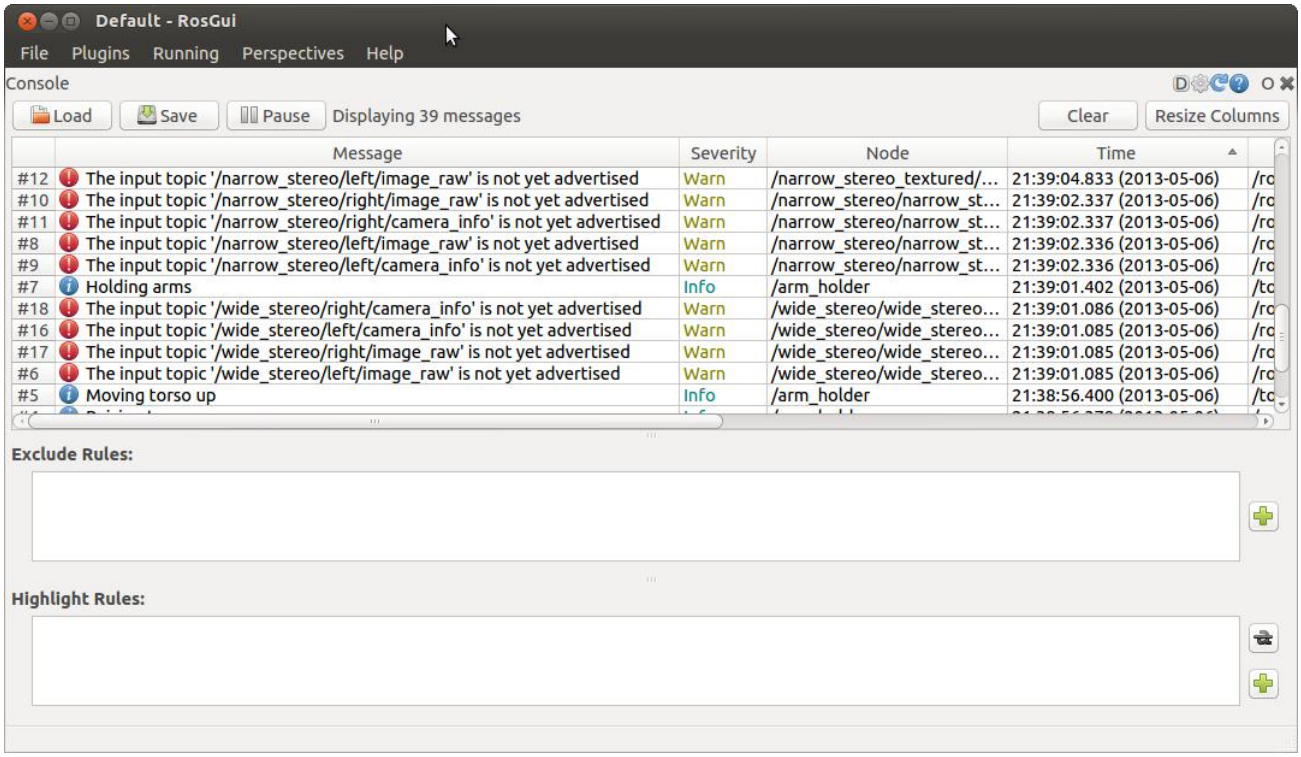


Image 26. rqt_console

You can invoke rqt_console by just typing:

rqt_console

The rqt_console window is divided into three subpanels:

i. The first panel outputs the logs. It has data about the message, severity/level, the node generating that message, and other data. Is here where you will extract all your logs data.
ii. The second one allows you to filter the messages issued on the first panel, excluding them based on criteria such as: node, severity level, or that it contains a certain word. To add a filter, just press the plus sign and select the desired one.

iii. The third panel allows you to highlight certain messages, while showing the other ones.

More information: http://wiki.ros.org/rqt_console

19.7 roswtf

roswtf is a tool for diagnosing issues with a running ROS system. You can either run it just by typing:

roswtf

roswtf looks for many, many things, and the list is always growing. There are two categories of what it looks for: file-system issues and online/graph issues[35]:

i. File-system issues: It checks enviromental variables, packages, and launch files, among other things. It looks for any inconsistencies that might be errors. You can use the command roswtf alone to get the system global status. But you can also use it to check particular launch files before using them. For that execute:

roscd <package>/launch
roswtf <launch file>

ii. Online/graph issues: roswtf also checks for any inconsistencies in the connections between nodes, topics, actions, and so on. It warns you if something is not connected or it's connected where it shouldn't be. These warnings aren't necessarily errors. They are simply things that ROS finds odd. It's up to you to know if it's an error or if it's just the way your project is wired.

```
Loaded plugin tf.tfwtf
================================================================================
Static checks summary:

Found 1 warning(s).
Warnings are things that may be just fine, but are sometimes at fault

WARNING ROS_HOSTNAME may be incorrect: ROS_HOSTNAME [192.168.2.23] resolves to [192.168.2.23], which does
not appear to be a local IP address ['127.0.0.1', '192.168.1.7'].

================================================================================

ROS Master does not appear to be running.
Online graph checks will not be run.
ROS_MASTER_URI is [http://192.168.2.2:11311]
```

Image 27. Example of roswtf execution

We can also run roswtf on launch files to search for potential issues:

roswtf <file_name>.launch

The wiki page of roswtf is available at http://wiki.ros.org/roswtf

19.8 Rosbag/rqt_bag

It is quite common in robotics that you want to record what a robot is doing and then use those recordings for analysis or for improving the performance of the algorithms, may be in a simulated environment. That recording is done via Bags[36].

A bag is a file format in ROS for storing ROS message data. Bags -- so named because of their .bag extension -- have an important role in ROS, and a variety of tools have been written to allow you to store, process, analyze, and visualize them.

Bags are typically created by a tool like rosbag, which subscribe to one or more ROS topics, and store the serialized message data in a file as it is received. These bag files can also be played back in ROS to the same topics they were recorded from, or even remapped to new topics.

Using bag files within a ROS Computation Graph is generally no different from having ROS nodes send the same data, though you can run into issues with timestamped data stored inside of message data. For this reason, the rosbag tool includes an option to publish a simulated clock that corresponds to the time the data was recorded in the file.

Some Bag tools to mention are:

i. rosbag: unified console tool for recording, playback, and other operations.
ii. rqt_bag: graphical tool for visualizing bag file data.
iii. rostopic: the echo and list commands are compatible with bag files.

The commands for playing with **rosbag** are:

To **Record** data from the topics you want:

rosbag record <topic1> <topic2>	*Will generate a file YYYY-MM-DD-HH-mm-ss.bag*
rosbag record -O <filename> <topic1>	*Will generate a file <filename>*
rosbag record -o <prefix> <topic1>	*Will generate a file <prefix>_YYYY-MM-DD-HH-mm-ss.bag*
rosbag record -a	*Will record all topics published*

To stop recording: ctrl+c

To get general information about the recorded data:

rosbag info name_bag_file.bag

To play:

rosbag play --clock <bag file>	*this will publish the topics as if it were the real time.*

The --clock flag will cause rosbag to publish the clock time from when the bag was recorded. If something else is also publishing time, such as the Gazebo simulator, this can cause a lot of problems. If two sources are publishing (different) times, then time will appear to jump around, and this will confuse the mapping algorithm (and possibly many other nodes). When you're using rosbag with the --clock-- argument, make sure that nothing else is publishing a time. The easiest way to do this is to stop any simulators you have running.

To use rosbag files, you have to make sure that the original data generator (real robot or simulation) is NOT publishing. Otherwise, you will get really weird data (the collision between the original and the recorded data). You have to also keep in mind that if you are reading from a rosbag, time is finite and cyclical, and therefore, you have to clear the plot area to view all of the time period.

To analyze a rosbag file we could use the command **rqt_bag**. Is is really useful in order to analyze the relationship between different topics in a time base.

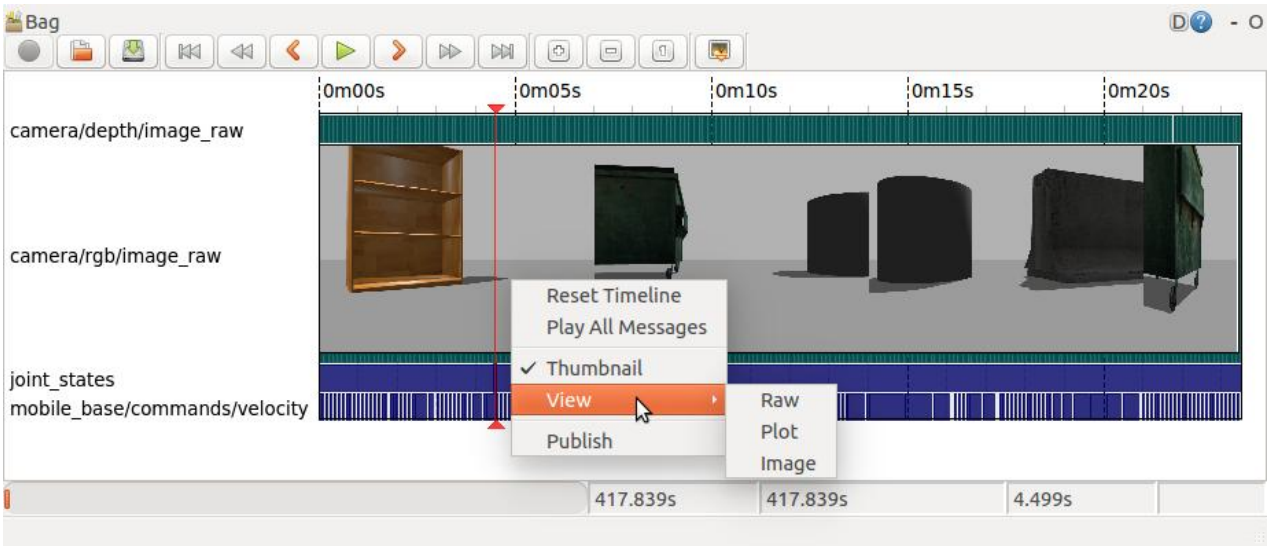


Image 28. rqt_bag example

More information about Rosbag and tools:

i. Rosbag command-line tool: http://wiki.ros.org/rosbag/Commandline
ii. Rqt_bag: http://wiki.ros.org/rqt_bag_plugins

19.9 tf Tools

19.9.1 view_frames

view_frames creates a diagram of the frames being broadcast by tf over ROS.

rosrun tf view_frames

Here a tf listener is listening to the frames that are being broadcast over ROS and drawing a tree of how the frames are connected, storing that in frames.pdf file. To view the tree:

evince frames.pdf

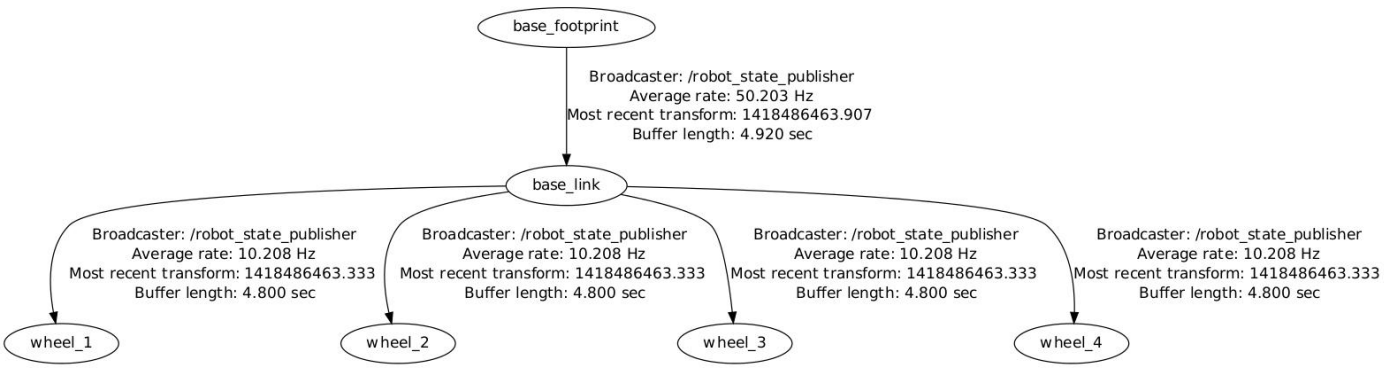


Image 29. Example of diagram created by view_frames

19.9.2 rqt_tf_tree

rqt_tf_tree is a runtime tool for visualizing the tree of frames being broadcast over ROS. You can refresh the tree simply by the refresh bottom in the top-left corner of the diagram.

rosrun rqt_tf_tree rqt_tf_tree

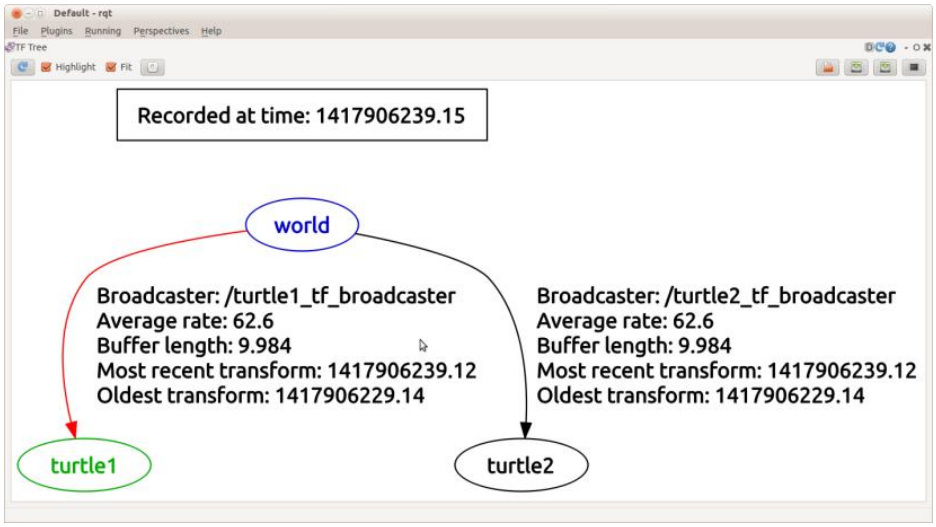


Image 30. Example of rqt_tf_tree

19.9.3 tf_echo

tf_echo reports the transform between any two frames broadcast over ROS.

rosrun tf tf_echo [reference_frame] [target_frame]

You will see the transform displayed as the tf_echo listener receives the frames broadcast over ROS. Example:

```
At time 1416409795.450
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.914, 0.405]
in RPY [0.000, -0.000, 2.308]
```

20 Appendix 0: Sensors configuration

This section explain about configuration of some common sensors. To understand in detail how to use messages from these sensors, go to the Messages section.

20.1 Depth camera setup

20.1.1 Kinect Installation procedure in turtlebot (Indigo)

Based on: http://edu.gaitech.hk/turtlebot/openKinect-turtlebot.html

First, you need to download the ROS OpenNI and OpenKinect (freenect) drivers for Indigo ROS (turtlebot) by running the following commands:

sudo apt-get install ros-indigo-openni- ros-indigo-openni2-* ros-indigo-freenect-**
rospack profile

Now you can test your camera. Type the following command:

export TURTLEBOT_3D_SENSOR=kinect
roslaunch freenect_launch freenect.launch *(Do not worry about the warnings and keep it running)*

To test the RGB image from camera type the following command:

rosrun image_view image_view image:=/camera/rgb/image_raw

To test the Mono image from camera type the following command:

rosrun image_view image_view image:=/camera/rgb/image_rect_mono

To test the depth image from camera type the following command (The darker the object is the closer it is to the turtlebot):

rosrun image_view image_view image:=/camera/depth/image_rect

Image 31. Example of depth image with kinect v2

20.1.2 Emulate a 2D laser scan from Depth image

This publish in /scan:

rosrun depthimage_to_laserscan depthimage_to_laserscan image:=/camera/depth/image_raw

More info on integrating in a launch file here:

How to use depthimage_to_laserscan package to create map?: https://answers.ros.org/question/246356/how-to-use-depthimage_to_laserscan-package-to-create-map/

https://answers.ros.org/question/192411/depthimage_to_laserscan-works-with-rosrun-but-the-depthimagetolaserscannodelet-does-not-work-properly/

Node depthimage_to_laserscan: http://wiki.ros.org/depthimage_to_laserscan

Consider that the stock files will have your kinect's virtual laser scan publishing on the /scan topic. This can be prevented by changing the topic that the Kinect laser publishes on with the following:

roscd turtlebot_bringup/launch/

edit 3dsensor.launch

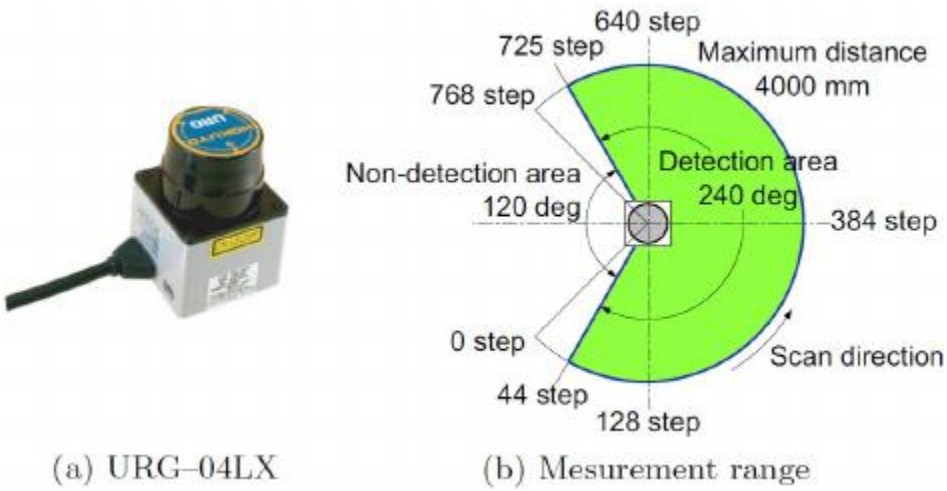
Look for the line that looks like this:

```
<arg name="scan_topic" default="scan"/>
```

Remove it and replace it with:

```
<arg name="scan_topic" default="kinect_scan"/>
```

20.2 Hokuyo Lidar setup



Install Hokuyo Node[37]

(Based on Using ROS to read data from a Hokuyo scanning laser rangefinder. Blake Hament. 2016. http://www.daslhub.org/unlv/wiki/doku.php?id=using_ros_to_read_data_from_a_hokuyo_scanning_laser_rangefinder)

ROS communicates with different sensors and motors through nodes. To download the hokuyo node, enter

sudo apt-get install ros-indigo-hokuyo-node

Plug in your hokuyo to the usb drive. At this point, you may be ready to go, but first let's confirm that the Hokuyo is properly connected and configured.

Check permissions with

ls -l /dev/ttyACM0

Your output should be in the form:

crw-rw-XX- 1 root dialout 166, 0 2016-09-12 14:18 /dev/ttyACM0

If XX is rw, then the laser is configured properly. If XX is –, then the laser is not configured properly and you need to change permissions like so

sudo chmod a+rw /dev/ttyACM0

Read data from Hokuyo sensor

At this point your Hokuyo sensor should be plugged into the usb port. To start ROS, enter

roscore

And then

rosrun hokuyo_node hokuyo_node

You should see the messages: “Connected to device” and shortly afterwards “Streaming data”. If you do not see these messages, check that the Hokuyo is being detected by your cpu by entering

lsusb -v

If you don’t see the Hokuyo listed anywhere, you may have a hardware issue.

Try using the following command to set the default port for the node.

rosparam set hokuyo_node/port /dev/ttyACM0

ROS uses “topics” to send and receive data between nodes. “Publishers” send data to a specific topic, and “subscribers” read the data published to a specific topic. In our case, the Hokuyo node will publish scan data to the /scan topic, and other nodes will subscribe to /scan to use that data for visualization or controls.

To check that the Hokuyo is publishing to /scan. Use

rostopic list

All active topics will be listed, check that /scan is present.
Next, check the messages being published to /scan by using

rostopic echo /scan

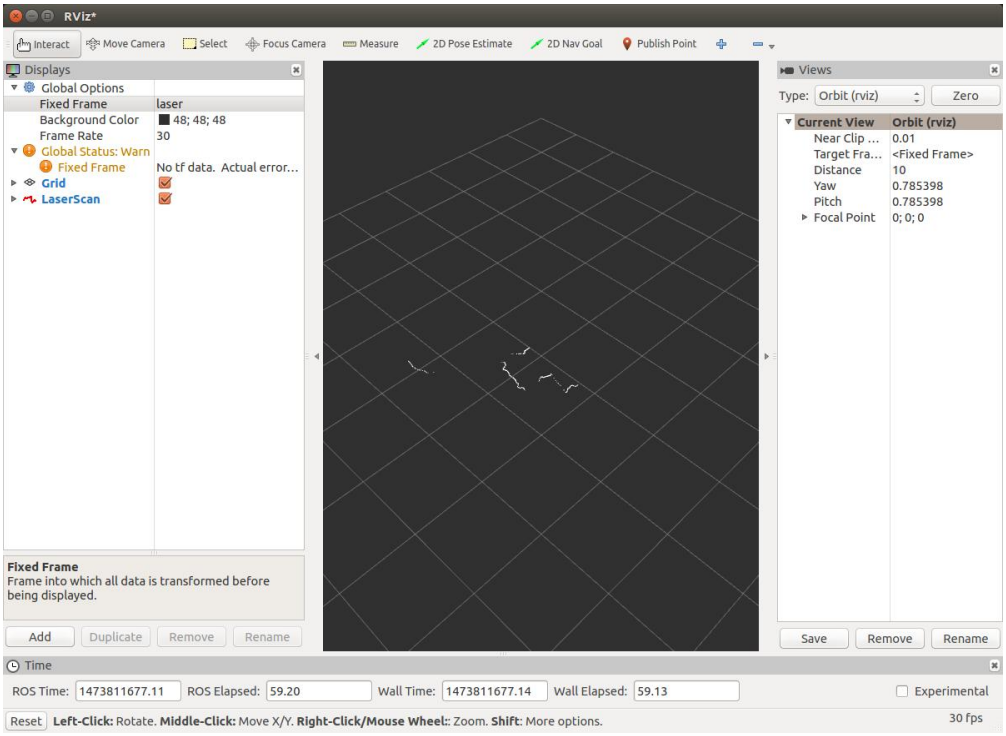
Visualize data

Hopefully you are now successfully streaming data from the Hokuyo. The final step in this tutorial is to visualize this data using rviz.

rosrun rviz rviz

Click add, then select the topic /scan. If you have an error related to tf, you need to manually enter your fixed frame as “/laser” in the textbox next to fixed frame on the left side of the GUI.

The final result should be a line mapping distances from the Hokuyo in a rectangular coordinate system.



20.3 RPLidar setup

RPLidar is a low cost Lidar manufactured by SlamTec (https://www.slamtec.com)

20.3.1 Specifications

360 degree scan field

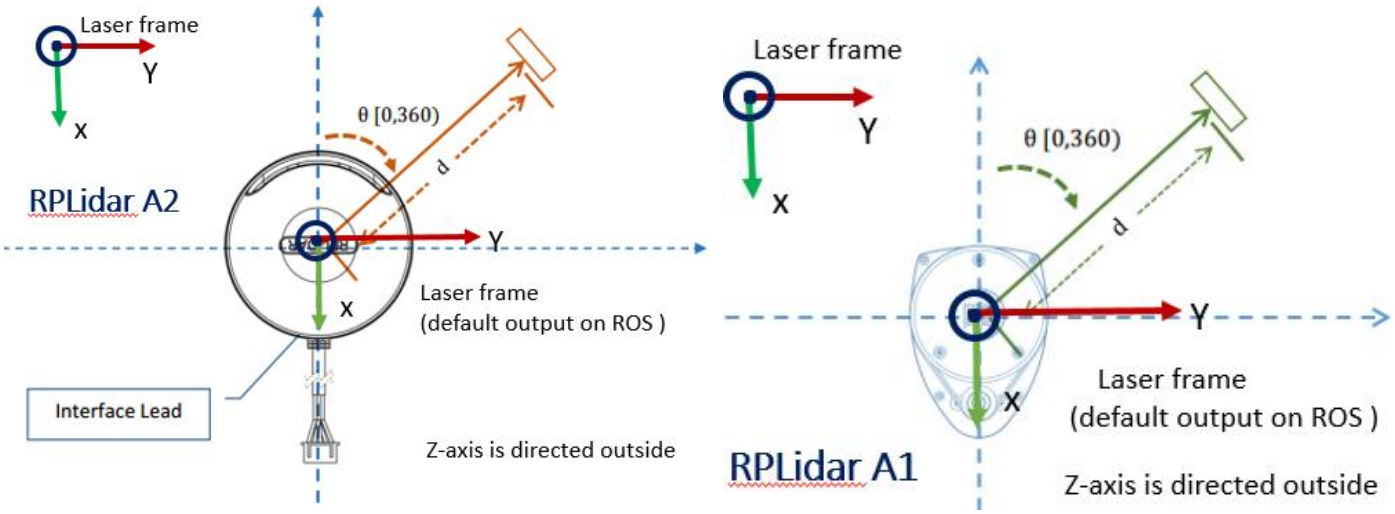
8 meter ranger distance

5.5hz/10hz rotating frequency (customizable 2hz to 10hz)

RPLIDAR A2 supports 4000 samples per second,

RPLIDAR A1 supports 2000 samples per second.

The driver publishes device-dependent sensor_msgs/LaserScan data



20.3.2 Installation

Information for installation and ROS implementation of RPLidar: http://wiki.ros.org/rplidar
ROS Rplidar: https://github.com/robopeak/rplidar_ros/wiki
Product webpage: https://www.slamtec.com/en/Lidar

Basic steps for installation:
Remember Turtlebot has indigo ROS version

cd ~/catkin_ws/src
git clone https://github.com/robopeak/rplidar_ros.wiki.git
cd ..
catkin_make or catkin build (if not installed catkin, execute: sudo apt-get install python-catkin-tools)
source devel/setup.bash

Now we could start rplidar node and view the scan result in rviz:

roslaunch rplidar_ros view_rplidar.launch (demo with Rviz)

If we want just to start rplidar node we execute:

roslaunch rplidar_ros rplidar.launch

and we can also run a client process to print raw scan at the screen:

rosrun rplidar_ros rplidarNodeClient

By default, the topic published is /scan, but we change it to /laserscan by modifying the launch file (rplidar_ros rplidar.launch) by adding. *<remap from="scan" to="laserscan"/>*

20.3.3 Persistent USB mapping

Any time we connect or disconnect the RPLidar it is assigned to a different USB port, so lets do it persistent: Some ideas here: http://hintshop.ludvig.co.nz/show/persistent-names-usb-serial-devices/ but summarized here:

Look for the VendorID:ProductID with Linux command lsusb with and without the Lidar connected. Compare and get the line difference, that correspond to the lidar. Example (in bold the Vendor ID:Product ID)

Bus 001 Device 020: ID **10c4:ea60** Cygnal Integrated Products, Inc. CP210x UART Bridge / myAVR mySmartUSB light

After checking with command

ls -l /dev | grep ttyUSB

we get that lidar is connected to ttyUSB1:

lrwxrwxrwx 1 root root 7 Oct 29 19:22 kobuki -> ttyUSB2
crw-rw---- 1 root dialout 188, 1 Oct 29 19:22 ttyUSB1
crw-rw-rw- 1 turtlebot dialout 188, 2 Oct 29 19:32 ttyUSB2

Then we need to find out the serial number by using:

udevadm info -a -n /dev/ttyUSB1 | grep '{serial}' | head -n1

We get: ATTRS{serial}=="0001"

Go to /etc/udev/rules.d. Create a new file called 99-usb-serial.rules (sudo gedit 99-usb-serial.rules) and put the following line in there:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="10c4", ATTRS{idProduct}=="ea60", ATTRS{serial}=="0001",
SYMLINK+="rplidar", MODE="0666"
```

Now the device names will continue to be assigned ad-hoc but the symbolic links will always point to the right device node. Let's see. Unplug rplidar and plug it back again. Then execute

ls -l /dev/rplidar

If we get the following, things are going ok:

lrwxrwxrwx 1 root root 7 Oct 29 19:45 /dev/rplidar -> ttyUSB1

Also we see:

ls -l /dev | grep ttyUSB

```
lrwxrwxrwx  1 root      root          7 Oct 29 19:45 kobuki -> ttyUSB2
lrwxrwxrwx  1 root      root          7 Oct 29 19:45 rplidar -> ttyUSB1
crw-rw----  1 root     dialout 188,   1 Oct 29 19:45 ttyUSB1
crw-rw-rw-  1 turtlebot dialout 188,   2 Oct 29  2017 ttyUSB2
```

The last step is to configure all the relevant tools to use these new names and forget about chasing the right /dev/ttyUSB* every second day.
Once you have change the USB port remap, you can change the launch file about the serial_port value.
<param name="serial_port" type="string" value="/dev/rplidar"/>

roscd rplidar_ros
cd launch
sudo gedit rplidar.launch

After editing should look like:

```
<launch>
<node name="rplidarNode"          pkg="rplidar_ros"  type="rplidarNode" output="screen">
<param name="serial_port"         type="string" value="/dev/rplidar"/>
<param name="serial_baudrate"     type="int"    value="115200"/>
<param name="frame_id"            type="string" value="laser"/>
<param name="inverted"            type="bool"   value="false"/>
<param name="angle_compensate"    type="bool"   value="true"/>
</node>
</launch>
```

20.3.4 Publishing frame into tf

An easy solution would be to manually starting a tf publisher that publishes the needed transform from the base_link (localized) to the laser scanner frame. To do that, add the following line to the file mentioned above ~/catkin_ws/src/rplidar_ros-master/launch/rplidar.launch

```
<node pkg="tf" type="static_transform_publisher" name="base_to_laser_broadcaster" args="0 0 0 0 0 0 base_link laser 100" />
```

Args are x y z orient. For an orientation of 180 degrees it is pi radians, so 3.14 (args="0 0 0 3.14 0 0)

21 Appendix 1: Project Mr. Messenger

21.1 Purpose

The purpose of this project is to configure and program a robot with ROS to go autonomously to specific locations on demand. A touch screen will show the locations. The user could put some weight on the robot and push the touchscreen for another destination to deliver the goods. This might be useful for home or office.

21.2 The hardware

We are using a Turtlebot 2 robot with Kobuki base38 as robot platform, a RPLidar 1 as Lidar, a Raspberry Pi 3B as embedded computer, a 7" touchscreen and a DCDC regulator 12v to 5v for powering the Raspberry Pi and Lidar from the robot batteries. The cost of these items is under 1,800 us$ (year 2019)

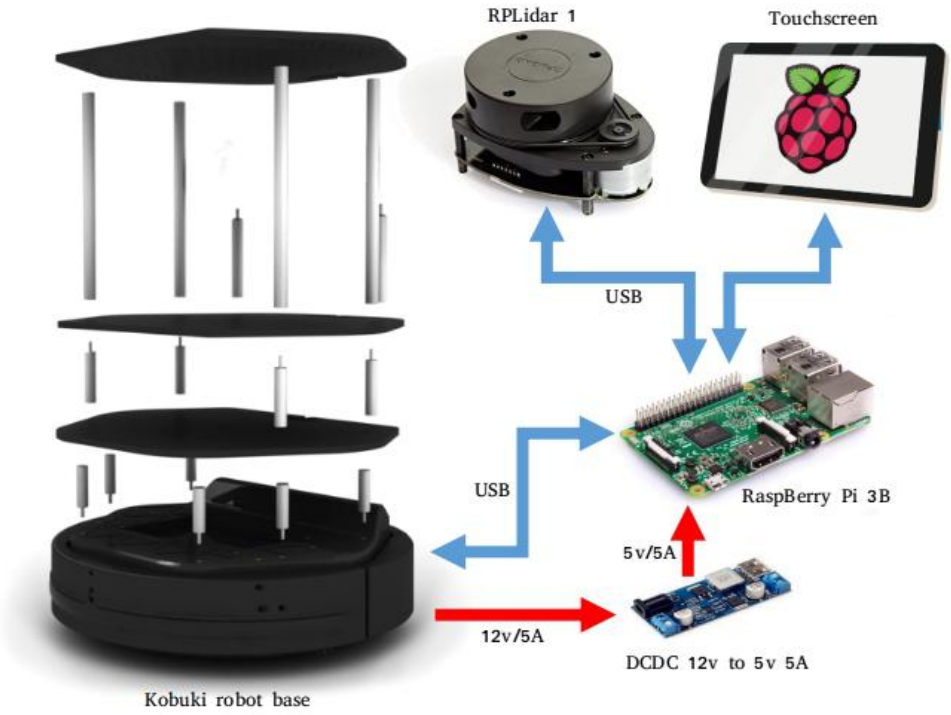


Image 32. Mr. Messenger

List of main components:

Component	Reference	Price	Link
Robot Platform	Turtlebot 2	1,100 us$	https://www.clearpathrobotics.com/turtlebot-2-open-source-robot/
Onboard computer	Raspberry Pi 3B	35 us$	https://www.adafruit.com/product/3055
SD card	16GB Ultra Micro SDHC UHS-I/Class 10 Card	9 us$	https://www.amazon.com/SanDisk-Ultra-Micro-Adapter-SDSQUNC-016G-GN6MA/dp/B010Q57SEE
Touch Screen	Daughter Board, Raspberry Pi 7" Touch Screen Display, 10 Finger Capacitive Touch	90 us$	https://www.raspberrypi.org/products/raspberry-pi-touch-display/
Lidar	RPLidar A1 (no longer available. A3 version is compatible)	450 us$	http://www.slamtec.com/en/lidar/a1
DCDC Regulator	D24V50F	15 us$	https://www.pololu.com/product/2851

21.3 The Software

The system uses ROS Kinetic over Ubuntu Mate Linux distribution on a Raspberry Pi 3B. See Appendix 2 at the end of this document for instructions on how to install Ubuntu Mate, ROS Kinetic, Turtlebot/Kobuki ROS drivers and RPLidar ROS drivers.

Here is the Flowchart of the application developed. All is under a package called mr_mess and is executed from the launch file mr_mess.launch by using the command roslaunch mr_mess mr_mess.launch

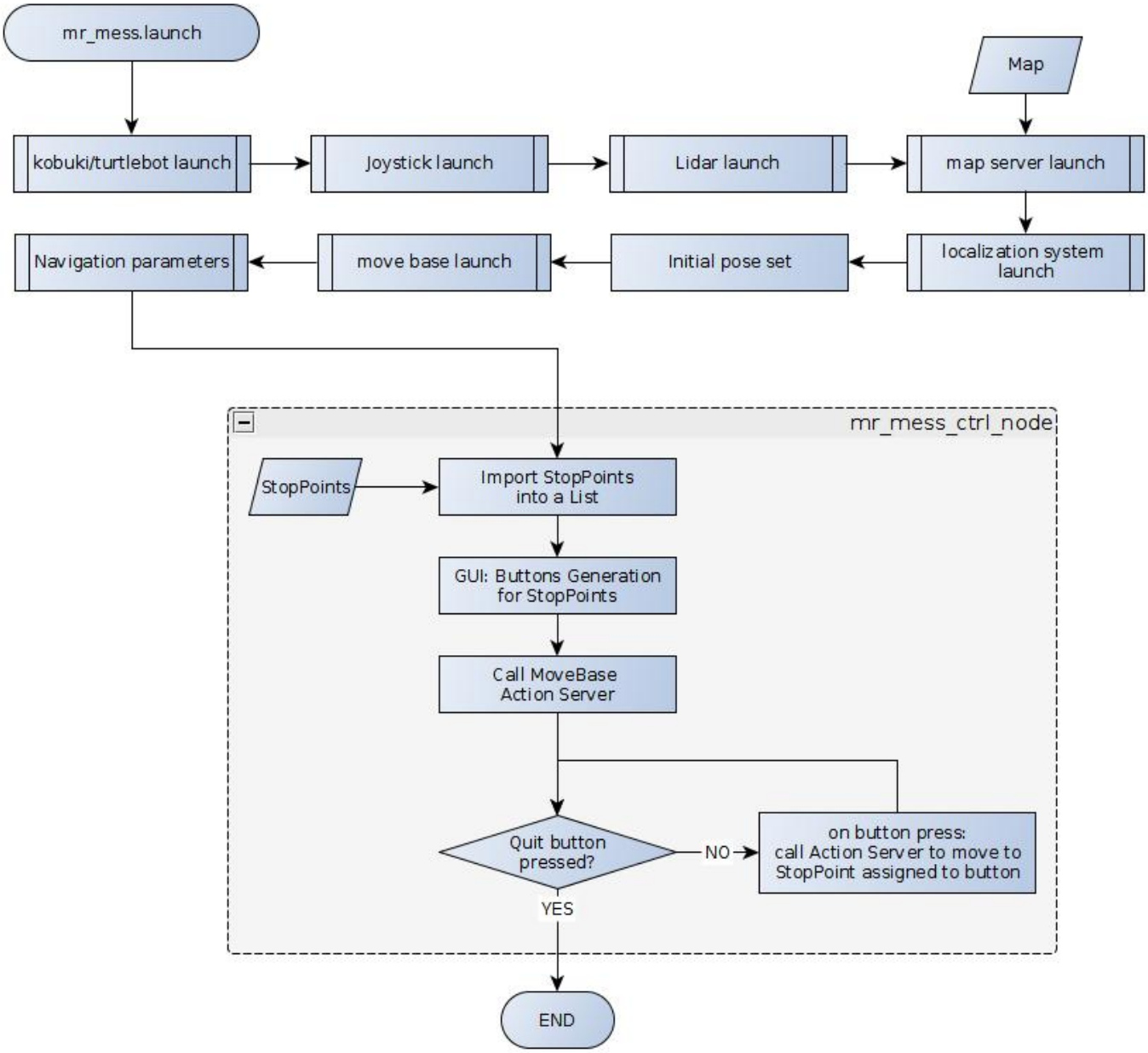


Image 33. Mr Messenger code flowchart

Let's see the different parts:

Kobuki/turtlebot launch

turtlebot_bringup provides roslaunch scripts for starting the TurtleBot base (kobuki) functionality. Inside the mr_mess.launch file it is called as:

```
<include file="$(find turtlebot_bringup)/launch/minimal.launch"/>
```

More information on this script: http://wiki.ros.org/turtlebot_bringup

Joystick launch

We configure our system to use a logitech remote Joystick for manual control of the robot. The manual control will always have priority over automated control. Inside the mr_mess.launch file it is called as:

```
<include file="$(find turtlebot_teleop)/launch/logitech.launch"/>
```

Lidar launch

This script launch the Lidar. In out case RPLidar v1. Inside the mr_mess.launch file it is called as:

```
<include file="$(find rplidar_ros)/launch/rplidar_setup2.launch"/>
```

Usually this launch file is called rplidar.launch, but as I use more than one robot with compatible RPLidars, I created one different Launch file per Lidar, with different frame.

For more information on this script, go to the Appendix 0, RPLidar setup.

map server launch

map_server provides the map_server ROS Node, which offers map data as a ROS Service. This map will be used by the navigation stack. Inside the mr_mess.launch file it is called as:

```
<node name="map_server" pkg="map_server" type="map_server" args="$(find mr_mess)/maps/map.yaml" />
```

map.yaml refers to the map used for the navigation. This map is created at the beginning of the system setup. See section 'How to create the map' further ahead.

More information on this script: http://wiki.ros.org/map_server

Localization system launch & Initial Pose set

AMCL uses a particle filter to track the position of the robot. The AMCL (Adaptive Monte Carlo Localization) package provides the amcl node, which uses the MCL system in order to track the localization of a robot moving in a 2D space. This node subscribes to the data of the laser, the laser-based map, and the transformations of the robot, and publishes its estimated position in the map. On startup, the amcl node initializes its particle filter according to the parameters provided. Inside the mr_mess.launch file it is called as:

```
<!-- Launch AMCL localization system & Initial Pose Set-->
<arg name="custom_amcl_launch_file" default="$(find
turtlebot_navigation)/launch/includes/amcl/amcl.launch.xml"/>
<arg name="initial_pose_x" default="6.1"/>
<arg name="initial_pose_y" default="9.7"/>
<arg name="initial_pose_a" default="3.14"/>
<include file="$(arg custom_amcl_launch_file)">
<arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
<arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
<arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
</include>
```

Previous code launch the localization system as well as define the pose (location and orientation) of the robot in the map, in the moment of running the system. Usually this is the location of the charging station. The location is defined in initial_pose_x and initial_pose_y and the orientation (in radians) in initial_pose_a. For more info on this, go to the section 'Identify key Stop_Points'

For more information on amcl, go to http://wiki.ros.org/amcl

move base launch

The move_base package provides an implementation of an action (see the actionlib package) that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner to accomplish its global navigation task.

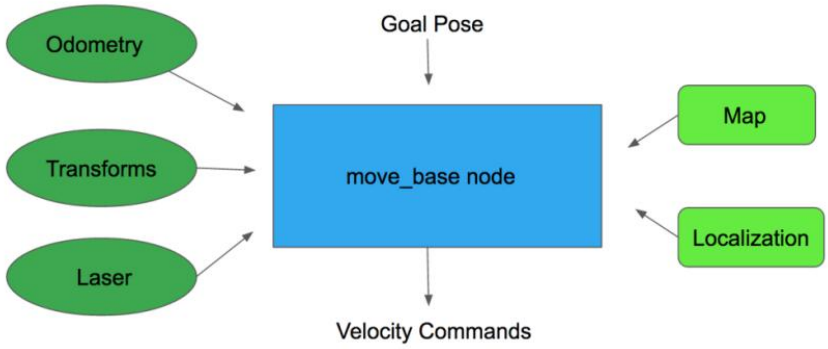


Image 34. move_base node

Inside the mr_mess.launch file it is called as:

```
<!-- Launch Move base server-->
<arg name="custom_param_file" default="$(find turtlebot_navigation)/param/r200_costmap_params.yaml"/>
<include file="$(find turtlebot_navigation)/launch/includes/move_base.launch.xml">
<arg name="custom_param_file" value="$(arg custom_param_file)"/>
</include>
```

More information: http://wiki.ros.org/move_base

Navigation parameters

This section set some navigation parameters defined in the mr_mess_nav_params.yaml file. Inside the mr_mess.launch file it is called as:

```
<!-- Parameters -->
<rosparam command="load" file="$(find mr_mess)/params/mr_mess_nav_params.yaml" />
```

These parameters are for the navigation stack as well as for the manual control with joystick, limiting velocities and accelerations:

```
#navigation parameters
/navigation_velocity_smoother/speed_lim_v: 0.2
/navigation_velocity_smoother/speed_lim_w: 0.4
```

```
/navigation_velocity_smoother/accel_lim_v: 0.3
/navigation_velocity_smoother/accel_lim_w: 0.4
/navigation_velocity_smoother/decel_factor: 0.5
/teleop_velocity_smoother/speed_lim_v: 0.2
/teleop_velocity_smoother/speed_lim_w: 0.4
/teleop_velocity_smoother/accel_lim_v: 0.3
/teleop_velocity_smoother/accel_lim_w: 0.4
/teleop_velocity_smoother/decel_factor: 0.5
```

mr_mess_ctrl_node

This is the interactive control program in Python. It shows a set of buttons in a touchscreen. Each button is associated to a location in the map (Stop_Points). When the user push one of the buttons, the robot goes to that location in the map using the ROS Navigation Stack. Inside the mr_mess.launch file it is called as:

```
<!-- Mr. Mess control launch file -->
<node pkg="mr_mess" type="mr_mess_ctrl.py" name="mr_mess_ctrl_node" output="screen">
</node>
```

The code is simple:

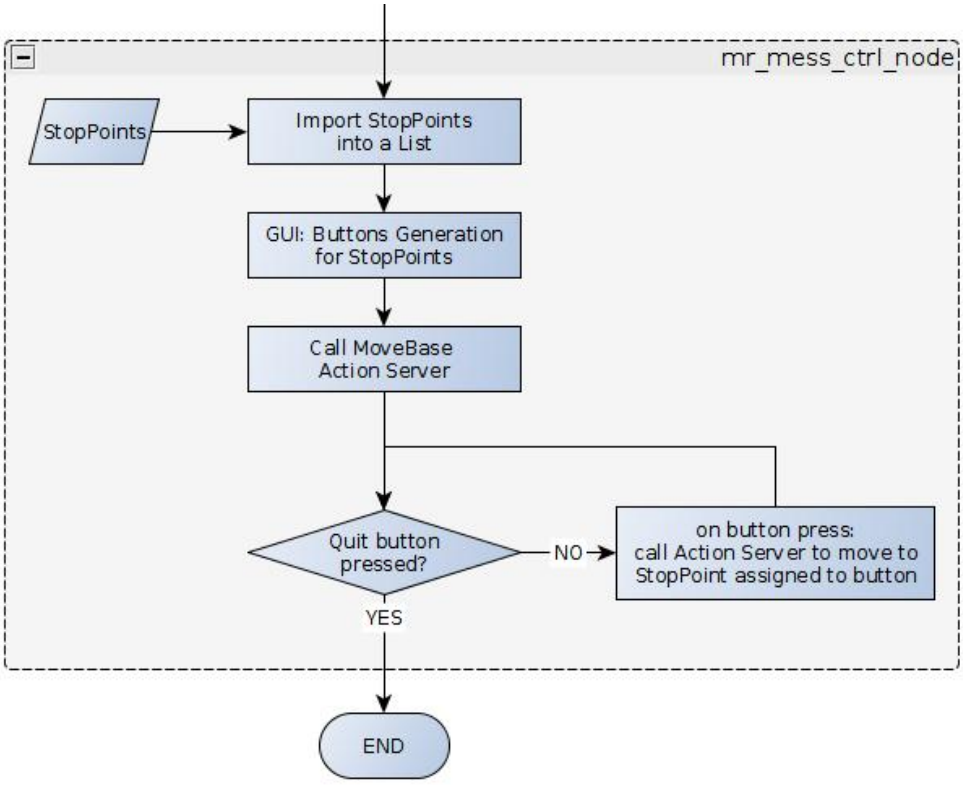


Image 35. mr_mess_ctrl_node flowchart

It loads the file stop_points.csv that contains data of the StopPoints: StopPoint_name, Position_X, Position_Y, Orientation_Z, Orientation_W

Then it generates one button for each of the StopPoints.

It calls de MoveBase action server and in case a button is pressed, the action server is called to move the robot to the pose set for the StopPoint.

For more details see the self explanatory code

21.4 How to create the map

First thing to have for a navigation system is a map. In this section we explain how to generate it and improve it. Basically we will have the Joystick ready and run in several terminal sessions the following commands:

roscore
source ~/catkin_ws/devel/setup.bash
rosparam set /robot_state_publisher/use_tf_static false
roslaunch turtlebot_bringup minimal.launch
roslaunch rplidar_ros rplidar_setup2.launch
roslaunch turtlebot_teleop logitech.launch
rosrun gmapping slam_gmapping

Use RViz to visualize the map building process:

roslaunch turtlebot_rviz_launchers view_navigation.launch

Move the robot around to create the map.

Finally, save the map to disk:

rosrun map_server map_saver -f map

it will generate one .pgm and one .yaml files. The first one has the bitmap image and the second one some info about the map.

21.5 Edit and improve the map

pgm image map has a lot of noise data. The image could be improved, cropped and rotated adequately. For that we could use Gimp (or any other image editor). Here we propose some hints using Gimp (https://www.gimp.org).

Some edits to do on map:

Rotate
Usually the map is rotated. We could try to align it with the field of view of the monitor.
Zoom the area of the map

Select Layer, Transform, Arbitrary Rotation and move the slider to rotate it accordingly.

Crop

Crop the map to the area with relevant information.

Use the Rectangle selection to select the area to crop. Then menu Image, Crop to selection.

Clear areas

There are many noisy points in areas that we know there are no obstacles.

Use the color picker tool to take the color of a clear area.

Use pencil tool and select thickness of tip. Then paint the areas not clear that should be. Is possible to do lines with shift. This helps specially to make aisles clear.

Redefine edges

Some times edges are not clearly defined in areas that should be. Use pencil tool and black color for them.

Forbidden areas

Paint a line where the robot should not go.

Bend an Image

Sometimes due to odometry errors or lack of odometry calibration, long aisle may appear bended in the maps and may be corrected by Gimp.: https://docs.gimp.org/en/plug-in-curve-bend.html

Then Save the map with Gimp format, just in case we need to do modifications in the future, and export as pgm. Ensure the extension is .pgm and saves as raw, not ascii

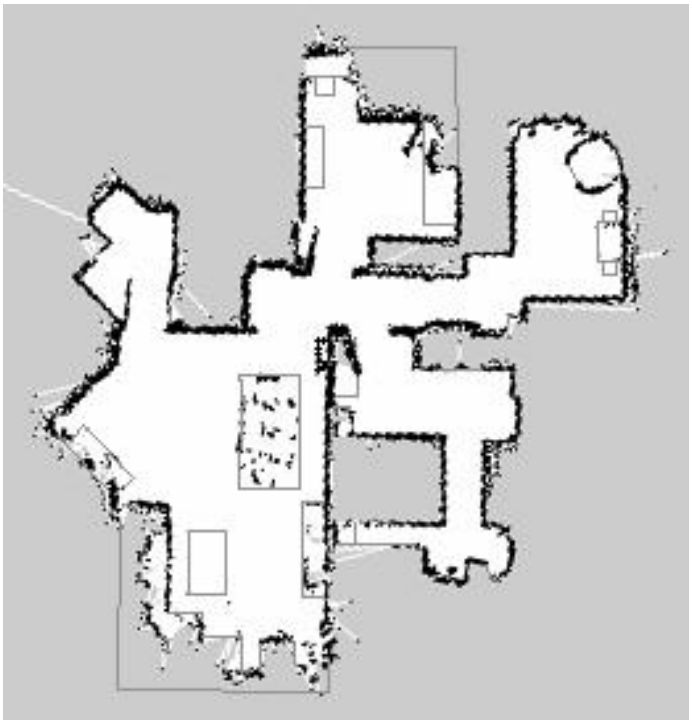


Image 36. Edited map

Now we need to modify the yaml file.

Edit the yaml file with any editor, like gedit. Modify the following data:

√ **image**: Name of the file containing the image of the generated Map. Ensure the name of the file is the one exported from Gimp.

√ **resolution**: Resolution of the map (in meters/pixel). Do not modify if you didn't change the size of the image. Just consider this value for the calculation of the next one.

√ **origin**: Coordinates of the lower-left pixel in the map. This coordinates are given in 2D (x,y) in meters, If it is ok to have the origin of the coordinates of the map at the lower-left corner, just set as 0,0. The third value indicates the rotation. If there's no rotation, the value will be 0. So the steps are to look for the properties of the file, size and calculate what is mentioned before.

example of map.yaml file:

```
image: map.pgm
resolution: 0.050000
origin: [0.000000, 0.000000, 0.000000]
```

```
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

after editing the map, leave it here: /home/turtlebot/catkin_ws/src/mr_mess/maps/

21.6 Identify key Stop Points

Stop Points are points of destination for the robot. We will ask the robot to go to any of those points, like the kitchen, the main door, the children room, the main room or the living room in case we are using the robot in a home, or the meeting room, the laboratory, the manager's office and so on in case of using in an office area. For this case we will consider a home

We will use an excel spreadsheet to collect the poses of the Stop Points. Here is an example of such spreadsheet:

Stop_Point	Position_X	Position_Y	Orientation_Z	Orientation_W
Dock Station	6.10	9.75	0.99	0.01
Entrance	2.05	6.01	0.92	0.37
Kitchen	0.37	8.66	0.69	0.71
Living room	3.61	4.07	-0.93	0.36
Dining room	4.22	4.23	0.36	0.92
Main room	7.09	5.84	-0.88	0.46
Kids room	10.39	8.63	0.60	0.79
Study room	6.68	10.97	0.71	0.70

To get the figures we will first launch the execute the following commands in several terminal sessions:

roslaunch turtlebot_bringup minimal.launch
roslaunch rplidar_ros rplidar_setup2.launch
roslaunch turtlebot_teleop logitech.launch
roslaunch turtlebot_navigation amcl_demo.launch map_file:=/home/turtlebot/catkin_/src/mr_mess/maps/map.yaml
roslaunch turtlebot_rviz_launchers view_navigation.launch

Use in Rviz the 2D pose Estimate button to identify the actual position and orientation of the robot.

If you have a joystick:

execute:

rostopic echo /amcl_pose

then move the robot using the joystick to the desired location. Then check for the last values shown in the terminal were you are running the rostopic echo. These are the figures you have to put in the excel spreadsheet.

If you do not have a joystick:

execute:

rostopic echo move_base_simple/goal

then use in Rviz the 2D Nav Goal button to set one by one the Stop Point poses. The robot will go to those positions. Move again the robot with this method until it is exactly located where you want. Then check for the last values shown in the terminal were you are running the rostopic echo. These are the figures you have to put in the excel spreadsheet

After filling up the excel spreadsheet, export as /home/turtlebot/catkin_/src/mr_mess/data/stop_points.csv

1.1 Future Works

Some future improvements for this basic version of the system would be:

i. Voice feedback. Always a robot that talks is a better user experience
ii. Obstacle avoidance
iii. Autonomous charging. It means going autonomously to a docking station for charging when the robot is idle or batteries are low

iv. Switch off lidar on demand. The lidar used is a low cost one. It is ok for the purpose of the project, although it is quite noisy. Switching it off when charging for example would be a good function to have
v. Come back to previous location if it could not reach the destination and show a message explaining the issue
vi. GUI for creation of the map, configuration of stop points (using joystick), spoken messages, strategies,...

22 Appendix 2: Install ROS Kinetic on Raspberry Pi for Turtlebot

Ubuntu Mate 16.04 installation[39]

Downloaded from here: https://ubuntu-mate.org/raspberry-pi/

Create SD bootable card in your computer. For that, install utilities:

sudo apt-get install gddrescue xz-utils
unxz ubuntu-mate-16.04.2-desktop-armhf-raspberry-pi.img.xz

Used disk utility to restore image to a 8GB SD card as in the video in previous link

Insert the SD card in RPi3 and boot[40]

During the initial configuration I give name turtlebot, computer's name 'turtlebot', user name 'turtlebot', password 'turtlebot', log in automatically

Raspberry Pi Resolution

execute in a terminal in Raspberry Pi:

sudo raspi-config

then you can choose Advanced Options then Resolution, you should be able to select optimal resolution settings. After rebooting the changes should take affect.

ROS installation

Then I install ROS following this link:

https://www.intorobotics.com/how-to-install-ros-kinetic-on-raspberry-pi-3-ubuntu-mate/

The only thing that changes is the catkin directory, that I used catkin_ws instead of catkin_workspace

sudo apt-get install python-rosinstall python-rosinstall-generator python-wstool build-essential

TURTLEBOT software

Install the Turtlebot drivers and tools software:

sudo apt-get install ros-kinetic-turtlebot-gazebo ros-kinetic-turtlebot ros-kinetic-turtlebot-apps ros-kinetic-turtlebot-interactions ros-kinetic-turtlebot-simulator

source /opt/ros/kinetic/setup.bash

Now we could connect to Turtlebot USB port and verify if it works:

roslaunch turtlebot_bringup minimal.launch
roslaunch turtlebot_teleop keyboard_teleop.launch or roslaunch turtlebot_teleop logitech.launch

and move with the joystick

RPI Lidar

Just do the following for installing the ROS drivers of RP Lidar

cd ~/catkin_ws/src
git clone https://github.com/robopeak/rplidar_ros.wiki.git
cd ..
catkin_make

Persistent USB mapping

Any time we connect or disconnect the RPLidar it is assigned to a different USB port, so lets do it persistent: Some ideas here:http://hintshop.ludvig.co.nz/show/persistent-names-usb-serial-devices/ but summarized here: Look for the VendorID:ProductID with Linux command lsusb with and without the Lidar connected. Compare and get the line difference, that correspond to the lidar. Example (in bold the Vendor ID:Product ID)

```
Bus 001 Device 020: ID 10c4:ea60 Cygnal Integrated Products, Inc. CP210x UART Bridge / myAVR mySmartUSB light
```

After checking with command

ls -l /dev | grep ttyUSB

we get that lidar is connected to ttyUSB1:

```
lrwxrwxrwx 1 root root 7 Oct 29 19:22 kobuki -> ttyUSB2
crw-rw---- 1 root dialout 188, 1 Oct 29 19:22 ttyUSB1
crw-rw-rw- 1 turtlebot dialout 188, 2 Oct 29 19:32 ttyUSB2
```

Then we need to find out the serial number by using:

udevadm info -a -n /dev/ttyUSB1 | grep '{serial}' | head -n1

We get: ATTRS{serial}=="0001"

Go to /etc/udev/rules.d. Create a new file called 99-usb-serial.rules (sudo gedit 99-usb-serial.rules) and put the following line in there:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="10c4", ATTRS{idProduct}=="ea60", ATTRS{serial}=="0001", SYMLINK+="rplidar", MODE="0666"
```

Now the device names will continue to be assigned ad-hoc but the symbolic links will always point to the right device node. Let's see. Unplug rplidar and plug it back again. Then execute

ls -l /dev/rplidar

If we get the following, things are going ok:

```
lrwxrwxrwx 1 root root 7 Oct 29 19:45 /dev/rplidar -> ttyUSB1
```

Also we see:

ls -l /dev | grep ttyUSB

```
lrwxrwxrwx 1 root root 7 Oct 29 19:45 kobuki -> ttyUSB2
lrwxrwxrwx 1 root root 7 Oct 29 19:45 rplidar -> ttyUSB1
crw-rw---- 1 root dialout 188, 1 Oct 29 19:45 ttyUSB1
crw-rw-rw- 1 turtlebot dialout 188, 2 Oct 29 2017 ttyUSB2
```

The last step is to configure all the relevant tools to use these new names and forget about chasing the right /dev/ttyUSB* every second day.

Once you have change the USB port remap, you can change the launch file about the serial_port value.
<param name="serial_port" type="string" value="/dev/rplidar"/>

roscd rplidar_ros
cd launch
sudo gedit rplidar.launch

After editing should look like:

```
<launch>
<node name="rplidarNode" pkg="rplidar_ros" type="rplidarNode" output="screen">
<param name="serial_port" type="string" value="/dev/rplidar"/>
<param name="serial_baudrate" type="int" value="115200"/>
<param name="frame_id" type="string" value="laser"/>
<param name="inverted" type="bool" value="false"/>
<param name="angle_compensate" type="bool" value="true"/>
</node>
```

```
</launch>
```

Publish RP Lidar frame into TF

It is necessary to publish the lidar frame referred to any other frame, An easy solution would be to manually starting a tf publisher that publishes the needed transform from the base_link (localized) to the laser scanner frame. To do that, add the following line to the file mentioned above ~/catkin_ws/src/rplidar_ros-master/launch/rplidar.launch

```
<node pkg="tf" type="static_transform_publisher" name="base_to_laser_broadcaster" args="0 0 0 0 0 0 base_link laser 100" />
```

Args are x y z orient. For an orientation of 180 degrees it is pi radians, so 3.14 (args="0 0 0 3.14 0 0)

SSH

Activate SSH on RPi by executing sudo service ssh start or raspi-config for permanent ssh activation
Then enter from another computer by using ssh turtlebot@turtlebot.local

Now configure SSH connections based on public keys:

First, generate public key and private key on local PC. Open a terminal and execute:

```
$ ssh-keygen
```

Press “Enter” to accept the default parameters, “id_rsa.pub” and “id_rsa”
will be generated in “~/.ssh” folder. Then copy “id_rsa.pub” to remote
computer:

$ scp ~/.ssh/id_rsa.pub turtlebot@ip_address:/home/turtlebot (“ip_addreee” is the IP address of remote computer)

After copy “id_rsa.pub” to remote computer, login on remote computer via SSH:

$ ssh turtlebot@ip_address

After login,append the content of “id_rsa.pub” to “~/.ssh/authorized_keys” on remote computer,and change the permission of “authorized_keys” file:

$ mkdir -p ~/.ssh/
$ cat id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 600 ~/.ssh/authorized_keys

Now you can login on remote computer without a password.

AUTOLOGIN

To configure the Desktop to auto-login add an autologin-user line specifying your user name to the /usr/share/lightdm/lightdm.conf.d/60-lightdm-gtk-greeter.conf file.

sudo /usr/share/lightdm/lightdm.conf.d/60-lightdm-gtk-greeter.conf
add: [SeatDefaults] greeter-session=lightdm-gtk-greeter autologin-user=turtlebot

Conf files in the /usr/share/lightdm/lightdm.conf.d/ directory are cascaded into the lightdm login manager in alpha order.

Redirect Audio Output

The sound will output to HDMI by default if both HDMI and the 3.5mm audio jack are connected. You can, however, force the system to output to a particular device using raspi-config.

For those of you who want to know how to do this without raspi-config:

For HDMI:

sudo amixer cset numid=3 2

For 3.5mm audio jack:

sudo amixer cset numid=3 1

Remote desktop

Two options: Share the existing X active desktop or create virtual instances. We need to install a VNC server and access it via Remmina on VNCviewer

Virtual instances:

Use vnc4server:
Follow instructions here: http://www.linuxeveryday.com/2017/08/install-configure-vnc-server-ubuntu-mate
Add to the xstartup configuration file mentioned in the link the following two lines:

```
unset SESSION_MANAGER
unset DBUS_SESSION_BUS_ADDRESS
```

Remote access to active desktop

We use teamviewer.
Access in RPi this link to download the server
app: https://download.teamviewer.com/download/linux/teamviewer-host_armhf.deb

cd Downloads
chmod +x teamviewer-host_armhf.deb
sudo dpkg -i teamviewer-host_armhf.deb

there will be several unmet dependencies. To fix this first type “sudo apt-get update”
“sudo apt-get -f install” to install all the dependencies or use “sudo apt-get -f upgrade” to install dependencies as well as upgrade other modules.
reboot
By default, the Teamviewer will start at the boot

Touchscreen display

If you get the official Raspberry Pi 7” touchscreen used in this project, you could install it using this tutorial: https://youtu.be/tK-w-wDvRTg

If after installing you see the screen inverted, just do the following:

open /boot/config.txt in your favourite editor and add the line:

```
lcd_rotate=2
```

This will rotate both the LCD and the touch coordinates back to the right rotation for our display stand.

Don't use the documented display_rotate, it performs a performance expensive rotation of the screen and does not rotate the touch input.

On-screen keyboard

Ubuntu Mate already comes with an on-screen keyboard. To get it go to the menu /Aplications /Universal Access /OnBoard

Backup/Restore SD card

It is good to keep a backup of the SD card and eventually restore it if needed. For than, follow this instructions if you have a Linux computer41:

Before inserting the SD card into the reader on your Linux PC, run the following command to find out which devices are currently available:

df -h

Which will return something like this:

```
Filesystem 1K-blocks Used Available Use% Mounted on
rootfs 29834204 15679020 12892692 55% /
/dev/root 29834204 15679020 12892692 55% /
devtmpfs 437856 0 437856 0% /dev
tmpfs 88432 284 88148 1% /run
tmpfs 5120 0 5120 0% /run/lock
tmpfs 176860 0 176860 0% /run/shm
/dev/mmcblk0p1 57288 14752 42536 26% /boot
```

Insert the SD card into a card reader and use the same df -h command to find out what is now available:

```
Filesystem 1K-blocks Used Available Use% Mounted on
rootfs 29834204 15679020 12892692 55% /
/dev/root 29834204 15679020 12892692 55% /
devtmpfs 437856 0 437856 0% /dev
tmpfs 88432 284 88148 1% /run
tmpfs 5120 0 5120 0% /run/lock
tmpfs 176860 0 176860 0% /run/shm
/dev/mmcblk0p1 57288 14752 42536 26% /boot
/dev/sda5 57288 9920 47368 18% /media/boot
/dev/sda6 6420000 2549088 3526652 42% /media/41cd5baa-7a62-4706-b8e8-02c43ccee8d9
```

The new device that wasn't there last time is your SD card.

The left column gives the device name of your SD card, and will look like '/dev/mmcblk0p1' or '/dev/sdb1'. The last part ('p1' or '1') is the partition number, but you want to use the whole SD card, so you need to remove that part from the name leaving '/dev/mmcblk0' or '/dev/sdb' as the disk you want to read from.

Open a terminal window and use the following to backup your SD card:

sudo dd if=/dev/sdb of=~/SDCardBackup.img

As on the Mac, the dd command does not show any feedback so you just need to wait until the command prompt re-appears.

To restore the image, do exactly the same again to discover which device is your SD card. As with the Mac, you need to unmount it first, but this time you need to use the partition number as well (the 'p1' or '1' after the device name). If there is more than one partition on the device, you will need to repeat the umount command for all partition numbers. For example, if the df -h shows that there are two partitions on the SD card, you will need to unmount both of them:

sudo umount /dev/sdb1
sudo umount /dev/sdb2

Now you are able to write the original image to the SD drive:

sudo dd bs=4M if=~/SDCardBackup.img of=/dev/sdb

The bs=4M option sets the 'block size' on the SD card to 4Meg. If you get any warnings, then change this to 1M instead, but that will take a little longer to write.

Again, wait while it completes. Before ejecting the SD card, make sure that your Linux PC has completed writing to it using the command:

sudo sync

23 Appendix 3: Install ROS Kinetic on Ubuntu 16.04 LTS

Needed Ubuntu 16.04 LTS
After installed Ubuntu, do in terminal the following:

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116

sudo apt-get update

sudo apt-get install ros-kinetic-desktop-full

sudo rosdep init

rosdep update

echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc

source ~/.bashrc

sudo apt-get install python-rosinstall python-rosinstall-generator python-wstool build-essential

mkdir -p ~/catkin_ws/src

cd catkin_workspace/src

catkin_init_workspace

sudo rosdep init

rosdep update

cd ~/catkin_ws/

catkin_make

source ~/catkin_ws/devel/setup.bash

echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc

make sure the stand-alone Gazebo works by running in terminal:

gazebo

Then go ahead with Turtlebot software installation, or other robot that you may use.

24 Appendix 4: TurtleBot

Technical Specifications: http://kobuki.yujinrobot.com/about2/

Tutorials:

i. https://www.clearpathrobotics.com/assets/guides/turtlebot/index.html
ii. http://edu.gaitech.hk/turtlebot/turtlebot-tutorials.html
iii. https://spectrum.ieee.org/tag/turtlebot%20tutorial

Considering you have ROS installed and you hace Turtlebot software installed (See next Appendix), then you can go on with the next steps.

This Launches Turtlebot. Execute in one terminal and leave there:

roslaunch turtlebot_bringup minimal.launch

This shows status of bumpers (in another terminal):

rostopic echo /mobile_base/events/bumper

In the result, the variable state can take the values 0 (RELEASED) and 1 (PRESSED). The variable bumper takes the values 0 (LEFT), 1 (CENTER), and 2 (RIGHT).

It is similar for Cliff sensor:

rostopic echo /mobile_base/events/cliff

Teleop:

i. Keyboard version: roslaunch turtlebot_teleop keyboard_teleop.launch
ii. Remote Joystick version: roslaunch turtlebot_teleop logitech.launch

SSH connection: *ssh turtlebot@turtlebot.local* and password turtlebot

Dashboard (Status of pc and AGV):

roslaunch turtlebot_dashboard turtlebot_dashboard.launch

Automatic docking: http://wiki.ros.org/kobuki/Tutorials/Automatic%20Docking

25 Appendix 5: Install Turtlebot software

After you have completed ROS instalation, then install turtlebot software:

sudo apt-get install ros-kinetic-turtlebot-gazebo ros-kinetic-turtlebot ros-kinetic-turtlebot-apps ros-kinetic-turtlebot-interactions ros-kinetic-turtlebot-simulator

source /opt/ros/kinetic/setup.bash

roslaunch turtlebot_gazebo turtlebot_world.launch

26 Appendix 6: Making Sounds (sound_play)

sound_play provides a ROS node that translates commands on a ROS topic (robotsound) into sounds. The node supports built-in sounds, playing OGG/WAV files, and doing speech synthesis via festival.

Details on the node: http://wiki.ros.org/sound_play

26.1 Installation

Considering Kinetic (Turtlebot):

sudo apt-get install ros-kinetic-sound-play
rosdep install sound_play
rosmake sound_play

26.2 Usage

To run it execute

rosrun sound_play soundplay_node.py

Then you could use different commands from the shell, like:

rosrun sound_play say.py "hello world"

More info here: http://wiki.ros.org/sound_play/Tutorials/ConfiguringAndUsingSpeakers

27 Appendix 7: Code snippets

27.1 Speed ramps

To use this snippet just store in LS_target, the linear speed (m/s) you want to reach and in Laccel the acceleration in m/s2 for the ramps. If this values are set from inside a callback function, remember to declare them as global inside it.

The snippet includes a function set_twist() and some additional code to include in the main loop of the python script.

```
#Constants and variables
Laccel=0.2              # Linear acceleration in m/s2
LS_target = 0           # Linear speed target

def set_twist():
# Set twist and publish on topic
# but take care to use the acceleration ramps
# LS_target global variable keep the desired value of linear speed
# Laccel global variable set the linear acceleration for the ramp

# Used physical formula for acceleration. Laccel= (Vf-Vi)/(Tf-Ti)
# therefore: Vf= Laccel*(Tf-Ti)+Vi

global Vi, Ti, Tf, LS_target

Tf = rospy.Time.now()

if LS_target>Vi:      Vf= Laccel*(Tf - Ti).to_sec() + Vi      # Increment on speed
if LS_target<Vi:      Vf=-Laccel*(Tf - Ti).to_sec() + Vi      # Decrement on speed
if LS_target==Vi:     Vf=Vi                                   # Speed reached

# Control speed limits
if (LS_target>=0) and (Vi<LS_target) and (Vf>LS_target): Vf=LS_target
if (LS_target<0)  and (Vi>LS_target) and (Vf<LS_target): Vf=LS_target
```

```
# Publish new value
move_cmd.linear.x = Vf
cmd_vel_.publish(move_cmd)

# Keep values for future iterations
Vi=Vf
Ti=Tf
```

---> To include in the main:

```
while not rospy.is_shutdown():
set_twist()                    # Publish Speed (linear and angular) with an acceleration ramp
rate.sleep()
```

27.2 Keyboard keys input

For keyboard input, we use a keyboard driver that listens for keystrokes and publishes them as std_msgs/String messages on the keys topic[42].

```
#!/usr/bin/env python
import sys, select, tty, termios
import rospy
from std_msgs.msg import String

if __name__ == '__main__':
    key_pub = rospy.Publisher('keys', String, queue_size=1)
    rospy.init_node("keyboard_driver")
    rate = rospy.Rate(100)
    old_attr = termios.tcgetattr(sys.stdin)
    tty.setcbreak(sys.stdin.fileno())
    print "Publishing keystrokes. Press Ctrl-C to exit..."
    while not rospy.is_shutdown():
        if select.select([sys.stdin], [], [], 0)[0] == [sys.stdin]:
            key_pub.publish(sys.stdin.read(1))
```

```
    rate.sleep()
termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_attr)
```

For subscribing we add to our main app:

```
rospy.Subscriber('keys', String, keys_cb)
```

And add the following callback. In this case check for the keys 0 to 6 and set the value of global variable ‘state’:

```
def keys_cb(msg):
# Keyboard input control
global state
key_mapping = { '0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6 }
if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
return # unknown key
state = key_mapping[msg.data[0]]
```

27.3 Navigate through waypoints

Example on how to make the robot navigate through different waypoints:

```
#!/usr/bin/env python
import rospy
import actionlib
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
waypoints = [
[(2.1, 2.2, 0.0), (0.0, 0.0, 0.0, 1.0)],
[(6.5, 4.43, 0.0), (0.0, 0.0, -0.984047240305, 0.177907360295)]
] #A list of the waypoints for the robot to patrol

def goal_pose(pose):
    # A helper function to turn a waypoint into a MoveBaseGoal
    goal_pose = MoveBaseGoal()
    goal_pose.target_pose.header.frame_id = 'map'
    goal_pose.target_pose.pose.position.x = pose[0][0]
```

```
    goal_pose.target_pose.pose.position.y = pose[0][1]
    goal_pose.target_pose.pose.position.z = pose[0][2]
    goal_pose.target_pose.pose.orientation.x = pose[1][0]
    goal_pose.target_pose.pose.orientation.y = pose[1][1]
    goal_pose.target_pose.pose.orientation.z = pose[1][2]
    goal_pose.target_pose.pose.orientation.w = pose[1][3]
    return goal_pose

if __name__ == '__main__':
    rospy.init_node('patrol')

    #Create a simple action client, and wait for the server to be ready
    client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    client.wait_for_server()

    while True:
        for pose in waypoints: #Loop through the waypoints, sending each as an action goal
        goal = goal_pose(pose)
        client.send_goal(goal)
        client.wait_for_result()
```

28 Appendix 8: Tricks

28.1 Some commands that helps

Rebuild dependencies:

sudo rosdep init
rosdep update

Make

cd ~/catkin_ws
catkin_make

Source

source ~/catkin_ws/devel/setup.bash

29 Bibliography

[1] ROS Wikipedia. https://en.wikipedia.org/wiki/Robot_Operating_System

[2] Ros.org (Wiki) http://wiki.ros.org/ROS/Introduction

[3] ROS in 5 days

[4] Understanding ROS Topics (Wiki): http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

[5] LaserScan Common Topics, Parameters, and Diagnostic Keys. Chad Rockey. 2013: http://www.ros.org/reps/rep-0138.html

[6] ROS common Messages (Wiki): http://wiki.ros.org/common_msgs

[7] Creating a ROS msg and srv (Wiki): http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv#Creating_a_msg

[8] slam_gmapping (Wiki): http://wiki.ros.org/slam_gmapping

[9] Creating a workspace for catkin (Wiki): http://wiki.ros.org/catkin/Tutorials/create_a_workspace

[10] ROS Packages (Wiki): http://wiki.ros.org/Packages

[11] ROS Nodes (Wiki): http://wiki.ros.org/Nodes

[12] Entrando a picar más profundo en ROS: https://brunofaundez.wordpress.com/tag/robotic-operating-system/

[13] Writing a Simple Publisher and Subscriber: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

[14] ROS Services (Wiki): http://wiki.ros.org/Services

[15] ROS Actions: https://index.ros.org/doc/ros2/Tutorials/Actions/

[16] ROS environment variables (Wiki): http://wiki.ros.org/ROS/EnvironmentVariables

[17] TurtleSim. Packt: https://subscription.packtpub.com/book/hardware_and_creative/9781788479592/1/ch01lvl1sec14/turtlesim-the-first-ros-robot-simulation

[18] Stage Robot Simulator: https://codedocs.xyz/CodeFinder2/Stage/md_README.html

[19] Adding Hokuyo Laser Finder to Turtlebot in Gazebo Simulation. Bharat Joshi. 2016: https://bharat-robotics.github.io/blog/adding-hokuyo-laser-to-turtlebot-in-gazebo-for-simulation/

[20] Coordinate Frames for Mobile Platforms. Wim Meeussen. https://www.ros.org/reps/rep-0105.html#coordinate-frames

[21] ROS Navigation Stack (Wiki): http://wiki.ros.org/navigation

[22] Odometry. Wikipedia. https://en.wikipedia.org/wiki/Odometry

[23] Publishing Odometry Information over ROS. Wiki. http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom

[24] SLAM for Dummies: An easy introduction to SLAM problem and solution. https://goo.gl/SrXuB3

[25] Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms: a technical and mathematical introduction to SLAM. https://goo.gl/MTB6Uw

[26] gmapping ROS package: http://wiki.ros.org/gmapping

[27] Map Building And Autonomous Navigation Using The Ros Navigation Stack And The Turtlebot. Matthew Stein. 2017. http://faculty.rwu.edu/mstein/verbiage/EMARO%20Mapping%20Report.pdf

[28] map_server ROS package: http://wiki.ros.org/map_server

[29] Costmaps. Roi Yehoshua. http://u.cs.biu.ac.il/~yehoshr1/89-685/Fall2013/ROSLesson5.pptx

[30] costmap_2d ROS package: http://wiki.ros.org/costmap_2d

[31] amcl ROS package: http://wiki.ros.org/amcl

[32] move_base ROS package: http://wiki.ros.org/move_base

[33] navfm ROS package: http://wiki.ros.org/navfn

[34] carrot_planner ROS package: http://wiki.ros.org/carrot_planner

[35] roswtf: http://wiki.ros.org/roswtf

[36] Bags. Wiki: http://wiki.ros.org/Bags

[37] Using ROS to read data from a Hokuyo scanning laser rangefinder. Blake Hament. 2016. http://www.daslhub.org/unlv/wiki/doku.php?id=using_ros_to_read_data_from_a_hokuyo_scanning_laser_rangefinder

[38] Kobuki user guide: https://docs.google.com/document/d/15k7UBnYY_GPmKzQCjzRGCW-4dIP7zl_R_7tWPLM0zKI/edit

[39] How To install ROS Kinetic on Raspberry Pi 3 (Ubuntu Mate): https://www.intorobotics.com/how-to-install-ros-kinetic-on-raspberry-pi-3-ubuntu-mate/

[40] Ubuntu MATE for the Raspberry Pi 2 and Raspberry Pi 3: https://ubuntu-mate.org/raspberry-pi/

[41] Backing up and Restoring your Raspberry Pi's SD Card: https://thepihut.com/blogs/raspberry-pi-tutorials/17789160-backing-up-and-restoring-your-raspberry-pis-sd-card

[42] Programming Robots with ROS. Morgan Quigley, Brian Gerkey, and William D. Smart. Published by O'Reilly Media, Inc.