Intelligent Robotics
Introduction to ROS

Introduction to ROS
ROS: Introduction

Robotics Software development before ROS:

- Lack of standards
- Little code reusability
- Keeping reinventing (or rewriting) device drivers, access to robot's interfaces, management of on-board processes, inter-process communication protocols, …
- Keeping re-coding standard algorithms
- New robot in the lab (or in the factory) →start re-coding (mostly) from scratch

Introduction to ROS
ROS: Introduction

ROS is an open-source **robot operating system**

- A set of software libraries and tools that help building robot applications that work across a wide variety of robotic platforms

- Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory and development at Willow Garage

- Since 2013 managed by OSRF(Open Source Robotics Foundation)

Introduction to ROS
ROS: Introduction

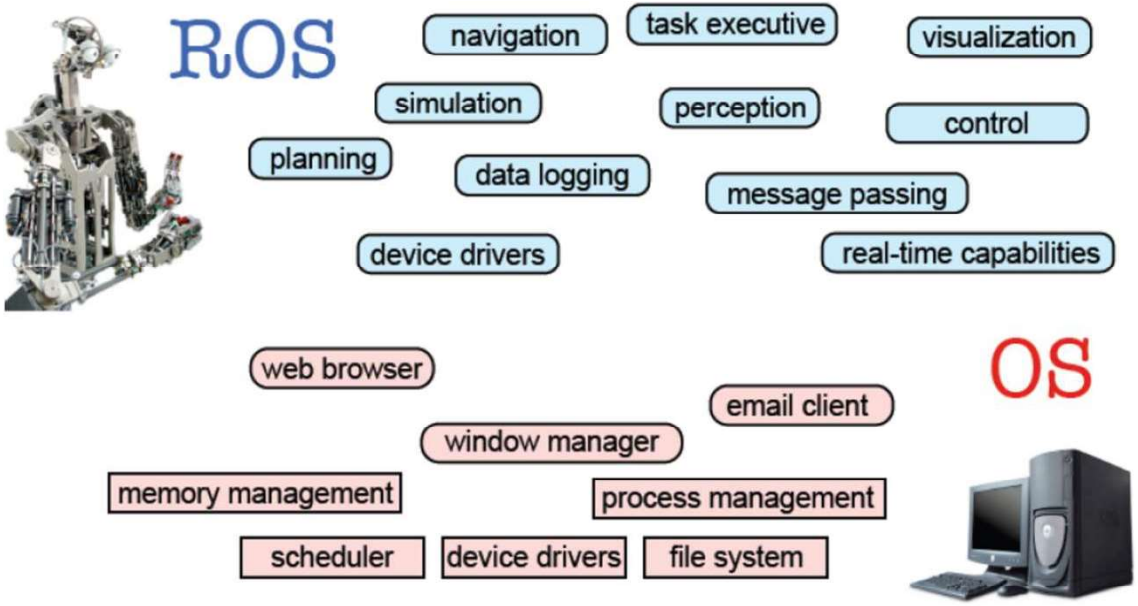
What is ROS?

- ROS (Robot Operative System) is an open-source, meta-operating system (or middleware) for robots. 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 provides tools and libraries for obtaining, building, writing, and running code across multiple computers.
- Code can be written in several languages like C++ or Python.

ROS mainly works on Linux!

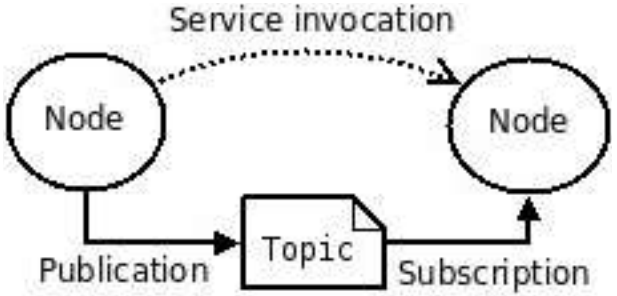
[http://wiki.ros.org]

Introduction to ROS
ROS: Introduction



Introduction to ROS
ROS: Key concepts

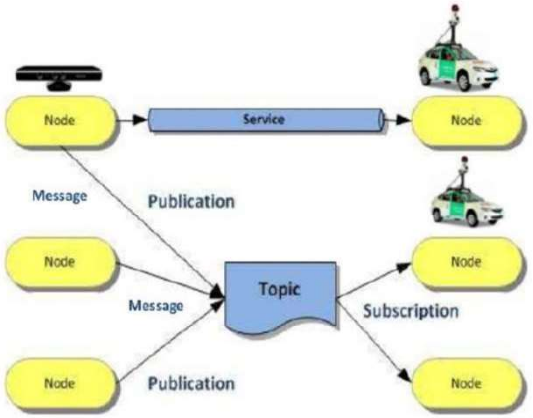
- **Nodes**: are processes that perform computation (programs). Nodes may control lasers or range-finders, joints or wheel motors, perform localization, path planning, and so on. Organized in **packages**.
- **Master**: provides name registration and lookup nodes. Without the Master, nodes would not be able to find each other or communicate (can be shared among different machines).
- **Communication**: is generally asynchronous (callbacks).
  - **Topics**: transport system with publish/subscribe
  - semantics (similar to pipes).
  - **Services**: One-shot requests.
  - **Actions**: One-shot requests, with control.

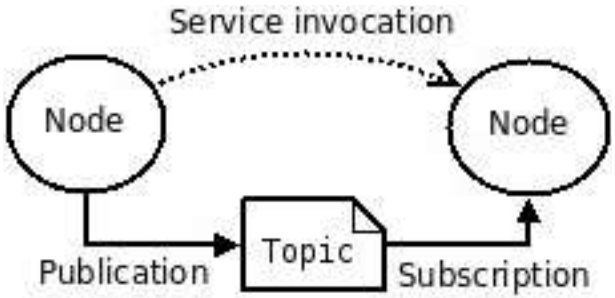


Introduction to ROS
ROS Nodes

Nodes:

- Single-purposed executable programs
  - e.g. sensor driver(s), actuator driver(s), map building, planner, UI, etc..
- Individually compiled, executed, and managed
- Nodes are written using aROS **client library**
  - roscpp–C++ client library
  - rospy–python client library
- Nodes can publish or subscribe to a **Topic**
- Nodes can also provide or use a **Service** or an **Action**





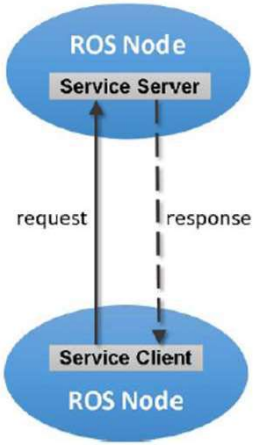
Introduction to ROS
ROS Master

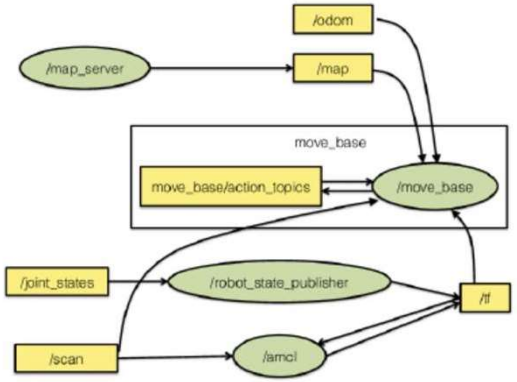
- Provides name registration and lookup nodes. Without the Master, nodes would not be able to find each other or communicate
- Each node connects to a Master to **register** details of the message streams they publish, services and actions that they provide, and streams, services, an actions to which that they to subscribe
- When a new node appears, the master provides it with the information that needs to form a direct peer-to-peer TCP-based connection with other nodes publishing and subscribing to the same message topics and services

Introduction to ROS
ROS Nodes

ROS Services:

- Synchronous inter-node transactions: ask for something and wait for it
- Service/Client model : 1 to 1 request response
- Service roles:
  - carry out remote computation
  - trigger functionality / behavior
  - map_server static_map
    retrieves the current grid map used for navigation



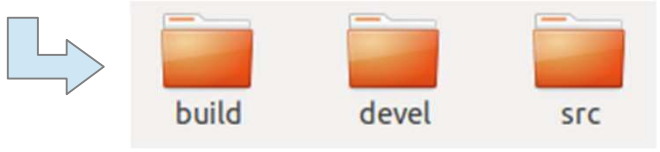


Introduction to ROS
ROS: organization and custom packages

Install and configure ROS:

http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

Workspace of ROS (catkin_ws or ros_ws)



Work here

A build system is responsible for generating 'targets' from raw source code that can be used by an end user

catkin is the official build system of ROS

It combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow

tail-shaped flower cluster found on willow trees

Introduction to ROS
ROS ROS Environment

- ROS is fully integrated in the Linux environment:
  - ROS_ROOT and ROS_PACKAGE_PATH
  - the ***rosbash*** package contains useful bash functions and adds tab-completion to a large number of ROS utilities

- After installing, ROS, setup.*sh files in '/opt/ros/<distro>/', need to be sourced to start *rosbash*: $ source /opt/ros/<distro>/setup.bash

- This command needs to be run on every new shell to have access to the ros commands: an easy way to do it is to add the line to the bash startup file (~/.bashrc)

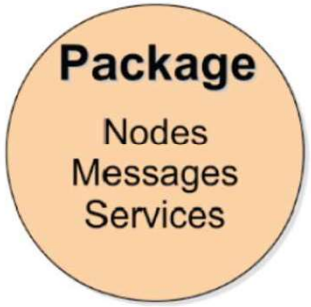
- Create catking workspace:
  - ```
    $ mkdir -p ~/catkin_ws/src
    $ cd ~/catkin_ws/
    $ catkin_make
    ```

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

Introduction to ROS
ROS ROS Package and Catkin Workspace

Software in ROS is organized in ***packages*.**

- A package contains one or more nodes, documentation, and provides a ROS interface
- Most of ROS packages are hosted in GitHub



Introduction to ROS
ROS Master

- ROS Master starts with $ roscore
- A node starts with $ rosrun package_name node_name
- List of nodes $rosnode list
- Info $rosnode info node_name
- $ rostopic list

Introduction to ROS
ROS ROS Package and Catkin Workspace

- Packages are the most atomic unit of build and the unit of release
- A package contains the source files for one node or more and configuration files
- A ROS package is a directory inside a **catkin workspace** that has a *package.xml* file **manifest** in it
- Manifest (package.xml): meta-information about a package (e.g., version, maintainer, license, etc.) and description of its dependencies (other ROS packages, messages, services, etc.). http://wiki.ros.org/catkin/package.xml
- A *catkin workspace* is a set of directories in which a set of related ROS code/packages live
- It's possible to have multiple workspaces, but only one-at-a-time can be used

Introduction to ROS
ROS ROS Package and Catkin Workspace

```
<package format="2">

 <name>foo_core</name>
 <version>1.2.4</version>

 <description>
   This package provides foo capability.
 </description>

 <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
 <license>BSD</license>

 <url>http://ros.org/wiki/foo_core</url>
 <author>Ivana Bildbotz</author>

 <buildtool_depend>catkin</buildtool_depend>

 <depend>roscpp</depend>
 <depend>std_msgs</depend>

 <build_depend>message_generation</build_depend>

 <exec_depend>message_runtime</exec_depend>
 <exec_depend>rospy</exec_depend>

 <test_depend>python-mock</test_depend>

 <doc_depend>doxygen</doc_depend>

</package>
```

tools which this package needs to build itself

which packages are needed to build this package

which packages are needed to run code in this package

additional dependencies for unit tests

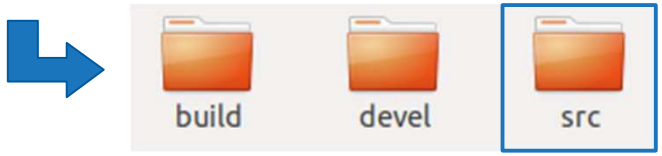
documentation tools which this package needs to generate documentation

Introduction to ROS
ROS: organization and custom packages

Install and configure ROS:

http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

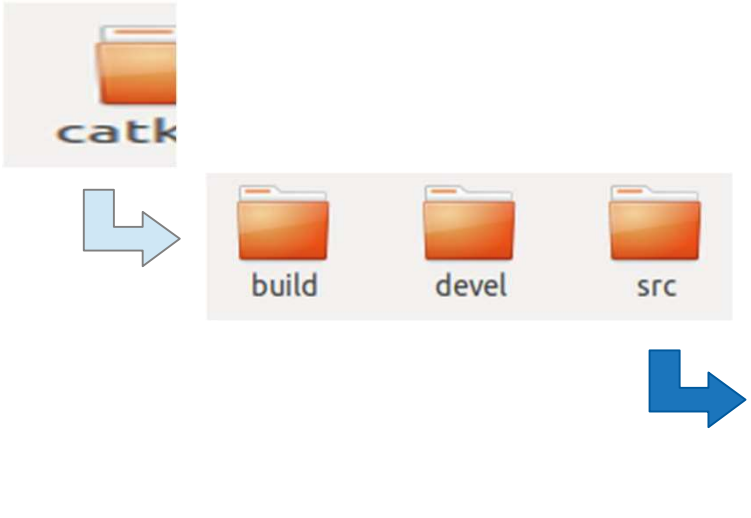
Workspace of ROS (catkin_ws or ros_ws)



Packages

Introduction to ROS
ROS: organization and custom packages

Create a custom ROS package:
http://wiki.ros.org/ROS/Tutorials/CreatingPackage

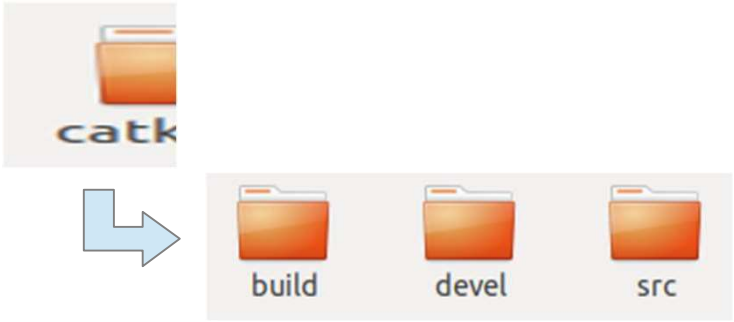



```
$ catkin_create_pkg my_first_pkg DEP1 DEP2
```

Introduction to ROS
ROS: organization and custom packages

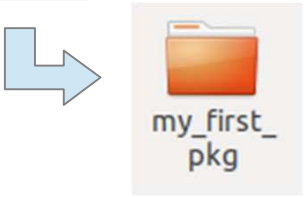
Create a ROS package:

http://wiki.ros.org/ROS/Tutorials/CreatingPackage



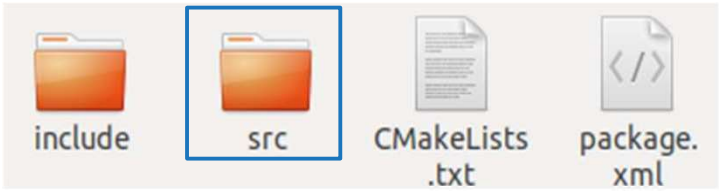
from src directory

```
$ catkin_create_pkg my_first_pkg DEP1 DEP2
```



src – source files
CMakeLists.txt – list of cmake rules
package.xml – package info and dep

Nodes source files



Introduction to ROS
ROS: organization and custom packages

Create a ROS package:

http://wiki.ros.org/ROS/Tutorials/CreatingPackage

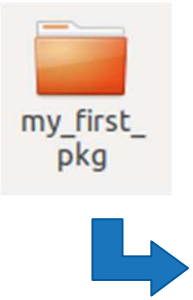

```
$ catkin_create_pkg my_first_pkg DEP1 DEP2
```

```
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

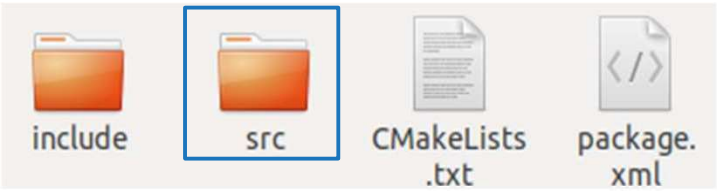
```
$ rospack depends1 beginner_tutorials
```

```
$ roscd beginner_tutorials
$ cat package.xml
```

```
<package>
 ...
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
 ...
</package>
```



Nodes source files



Introduction to ROS
ROS ROS Package and Catkin Workspace

- Layout of the *src/my_package* folder in a catkin workspace:

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

- Source files implement nodes, can be written in multiple languages
- Nodes are launched individually or in groups, using *launch files*

Introduction to ROS
ROS ROS Topics and Messages

```
$ roscore
```

```
$ rosrun turtlesim turtlesim_node
```

```
$ rosrun turtlesim turtle_teleop_key
```

```
$ sudo apt-get install ros-<distro>-rqt
$ sudo apt-get install ros-<distro>-rqt-common-plugins
```

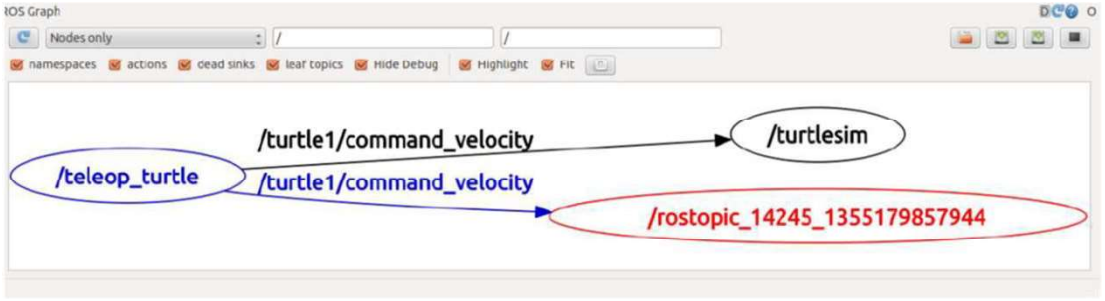
```
$ rosrun rqt_graph rqt_graph
```


```
$ rostopic echo /turtle1/command_velocity
```

```
$ rostopic type /turtle1/command_velocity
```

```
$ rosmsg show turtlesim/Velocity
```

```
float32 linear
float32 angular
```



Introduction to ROS
ROS ROS Messeges

```
$ roscore
```

```
$ rosrun turtlesim turtlesim_node
```

```
$ rosrun turtlesim turtle_teleop_key
```

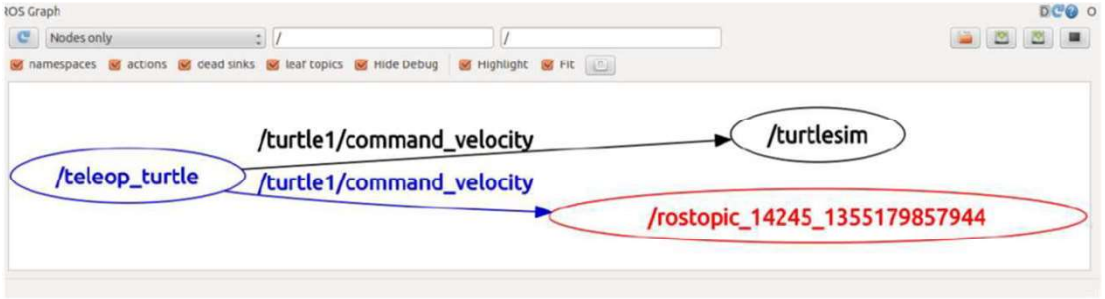
```
$ sudo apt-get install ros-<distro>-rqt
$ sudo apt-get install ros-<distro>-rqt-common-plugins
```

```
$ rosrun rqt_graph rqt_graph
```


```
$ rosmsg show turtlesim/Velocity
```

```
$ rosmsg show geometry_msgs/Twist
```

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



Introduction to ROS
ROS ROS Messeges

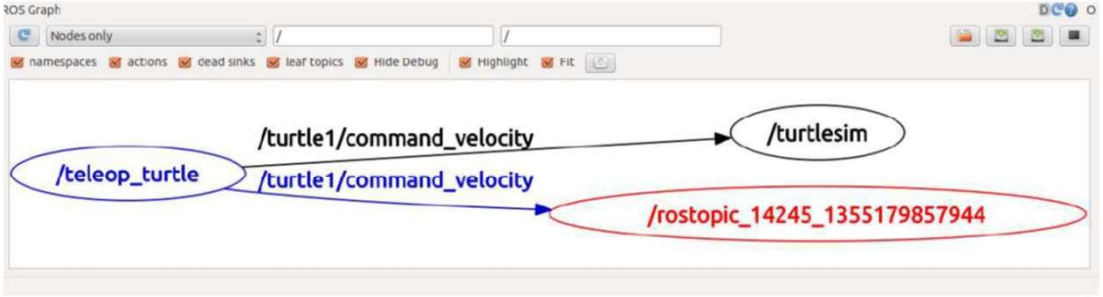
```
$ roscore
```

```
$ rosrun turtlesim turtlesim_node
```

```
$ rosrun turtlesim turtle_teleop_key
```

```
$ sudo apt-get install ros-<distro>-rqt
$ sudo apt-get install ros-<distro>-rqt-common-plugins
```

```
$ rosrun rqt_graph rqt_graph
```




```
$ rostopic pub -1 /turtle1/command_velocity turtlesim/Velocity -- 2.0 1.8
```

```
$ rostopic pub /turtle1/command_velocity turtlesim/Velocity -r 1 -- 2.0 -1.8
```

```
$ rostopic hz /turtle1/pose
```

Introduction to ROS
ROS ROS Services and Parameters

```
$ roscore
```

```
$ rosrun turtlesim turtlesim_node
```

```
$ rosrun turtlesim turtle_teleop_key
```

```
$ sudo apt-get install ros-<distro>-rqt
$ sudo apt-get install ros-<distro>-rqt-common-plugins
```

```
$ rosrun rqt_graph rqt_graph
```

```
$ rosservice list
```

```
$ rosservice call clear
```

```
$ rosservice call spawn 2 2 0.2 ""
```

```
$ rosparam list
```

```
$ rosparam get /
```

```
$ rosparam set background_r 150
```

Introduction to ROS
ROS ROS msg and srv

msg: msg files are simple text files that describe the fields of a ROS message

srv: srv file describes a service; two parts, a request and a response.

- •int8, int16, int32, int64 (plus uint*)
- •float32, float64
- •string
- •time, duration
- •other msg files
- •variable-length array[] and fixed-length array[C]

special type ROS: Header

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

```
int64 A int64 B
---
int64 Sum
```

In package.xml

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

In CMakeLists.txt

```
$ roscd beginner_tutorials
$ mkdir msg
$ echo "int64 num" > msg/Num.msg
```

```
string first_name
string last_name
uint8 age
uint32 score
```

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

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

```
generate_messages(
 DEPENDENCIES
 std_msgs )
```

```
catkin_package(
...
CATKIN_DEPENDS message_runtime ...
...)
```

make the package again

```
$ roscd beginner_tutorials
$ cd ../..
$ catkin_make $ cd -
```

Introduction to ROS
ROS: node example (C++)

Publisher/subscriber example:
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

Initialize ROS and enable publisher

```
#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);
  while (ros::ok()) {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();
    ROS_INFO("%s", msg.data.c_str());
   chatter_pub.publish(msg);
   ros::spinOnce();
   loop_rate.sleep();
   ++count;
   }
  return 0;
  }
```

- Initialize the ROS system
- Advertise that we are going to be publishing std_msgs/String messages on the chatter topic to the master
- Loop while publishing messages to chatter 10 times a second

Initialize ROS. Node names must be base name and unique in a running system.

Create a handle to this process' node.
The first NodeHandle created do the initialization of the node

ros::Publisher object, which serves two purposes: 1) it contains a publish() method that lets you publish messages onto the topic it was created with, and 2) when it goes out of scope, it will automatically unadvertise.

Publish into the main loop. The message can be received asynchronously

Introduction to ROS
ROS: node example (C++)

Publisher/subscriber example:
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

Initialize ROS and enable publisher and subscriber (the latter with callback)

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
 ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "listener");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
    ros::spin();

    return 0;
}
```

Callback function which is called when a new message has arrived on the chatter topic.

Subscribe to the chatter topic with the master.
ROS will call the chatterCallback() function whenever a new message arrives.
The 2nd argument is the queue size

ros::spin() enters a loop, calling message callbacks as fast as possible.

Introduction to ROS
ROS: node example (C++)

Publisher/subscriber example:

http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

add these lines to the bottom of your CMakeLists.txt:

```
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener beginner_tutorials_generate_messages_cpp)
```

talker and listener by default will go into package directory of your devel space, located by default at ~/catkin_ws/devel/lib/<package name>.

Now run catkin_make:

```
# In your catkin workspace
$ cd ~/catkin_ws
$ catkin_make
```

```
# In your catkin workspace
$ cd ~/catkin_ws
$ source ./devel/setup.bash
```

running the Publisher and the listener

```
$ roscore
```

```
$ rosrun beginner_tutorials talker (C++)
```

```
$ rosrun beginner_tutorials listener (C++)
```

Introduction to ROS
ROS: node example (C++)

Publisher/subscriber example:

http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

add these lines to the bottom of your CMakeLists.txt:

```
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener beginner_tutorials_generate_messages_cpp)
```

talker and listener by default will go into package directory of your devel space, located by default at ~/catkin_ws/devel/lib/<package name>.

Now run catkin_make:

```
# In your catkin workspace
$ cd ~/catkin_ws
$ catkin_make
```

```
# In your catkin workspace
$ cd ~/catkin_ws
$ source ./devel/setup.bash
```

running the Publisher and the listener

```
$ roscore
```

```
$ roslaunch beginner_tutorials talk-list.launch
```

Introduction to ROS
ROS: node example (C++)

Publisher/subscriber example:

http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

running the Publisher and the listener

```
$ roscore
```

```
$ roslaunch beginner_tutorials talk-list.launch
```

Nella cartella launch il file talk-listen.launch

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
 <node name="talker" pkg ="beginner_tutorials"  type="talker" output="screen" />
 <node name="listener" pkg ="beginner_tutorials" type="talker" output="screen" />
</launch>
```

Introduction to ROS
ROS: useful commands (nodes)

`$ roscore`	Starts the roscore, this have to be always running.
`$ rosrun PKG_NAME NODE_NAME`	Starts a ROS node.
`$ roslaunch PKG_NAME LAUNCH_FILE_NAME`	Starts a .launch file. This allows multiple nodes to be started.
`$ rosnode list`	Plots the list of all nodes running.
`$ catkin_make`	Compiles all C++ nodes and packages.

Introduction to ROS
ROS: useful commands (nodes)

`$ roscore`	Starts the roscore, this have to be always running.
`$ rosrun PKG_NAME NODE_NAME`	Starts a ROS node.
`$ roslaunch PKG_NAME LAUNCH_FILE_NAME`	Starts a .launch file. This allows multiple nodes to be started.
`$ rosnode list`	Plots the list of all nodes running.
`$ catkin_make -DCATKIN_WHITELIST_PACKAGES="PKG_NAME"`	Compiles a single package

Introduction to ROS
ROS: useful commands (topics)

`$ rostopic list`	Lists all topics published or subscribed by running nodes.
`$ rostopic info TOPIC_NAME`	Plots the info of a topic like type or publishing subscribing nodes.
`$ rostopic echo TOPIC_NAME`	Plots the value of a topic in real-time.
`$ rostopic pub TOPIC_NAME TYPE VALUE`	Publish a value on a topic.
`$ rviz`	Graphical visualization of topics.

Introduction to ROS
ROS: packages from repository

ROS have a huge community that continuously develop packages and nodes (often implementing state-of-the-art algorithms).

There are thousands of packages and nodes!

```
$ sudo apt-get install ros-DISTRO-PKG_NAME
```

(e.g.)

```
$ sudo apt-get install ros-melodic-amcl
```

DISTRO: Melodic
PKG: Adaptive Monte Carlo Localization

Introduction to ROS
ROS: Turtlebot

http://wiki.ros.org/turtlebot/Tutorials/indigo/Turtlebot%20Installation

```
sudo apt-get install ros-indigo-turtlebot ros-indigo-turtlebot-apps ros-indigo-turtlebot-interactions
ros-indigo-turtlebot-simulator ros-indigo-kobuki-ftdi ros-indigo-rocon-remocon ros-indigo-rocon-qt-library
ros-indigo-ar-track-alvar-msgs
```

Indigo version

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

Turtlebot in the gazebo environment

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Turtlebot in teleoperation

```
rosrun rqt_graph rqt_graph
```

Introduction to ROS
ROS: Gazebo

Robot simulation

Gazebo simulates populations of robots in complex indoor and outdoor environments. It provides physics engine, rendering, sensor generation, and graphical interfaces.

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

The client and server communicate using the gazebo communication library.

World Files

The world description file contains all the elements in a simulation, including robots, lights, sensors, and static objects. This file is formatted using SDF (Simulation Description Format), and typically has a .world extension.The Gazebo server (gzserver) reads this file to generate and populate a world.

Model Files

A model file uses the same SDF format as world files, but should only contain a single <model> ... </model>. The purpose of these files is to facilitate model reuse, and simplify world files.

Environment Variables

Gazebo uses a number of environment variables to locate files, and set up communications between the server and clients. Default values that work for most cases are compiled in.

Introduction to ROS
ROS: Turtlebot

http://wiki.ros.org/turtlebot/Tutorials/indigo/Turtlebot%20Installation

```
sudo apt-get install ros-indigo-turtlebot ros-indigo-turtlebot-apps ros-indigo-turtlebot-interactions
ros-indigo-turtlebot-simulator ros-indigo-kobuki-ftdi ros-indigo-rocon-remocon ros-indigo-rocon-qt-library
ros-indigo-ar-track-alvar-msgs
```

Indigo version

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

Turtlebot in the gazebo environment

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Turtlebot in teleoperation

```
roslaunch turtlebot_rviz_launchers view_robot.launch
```

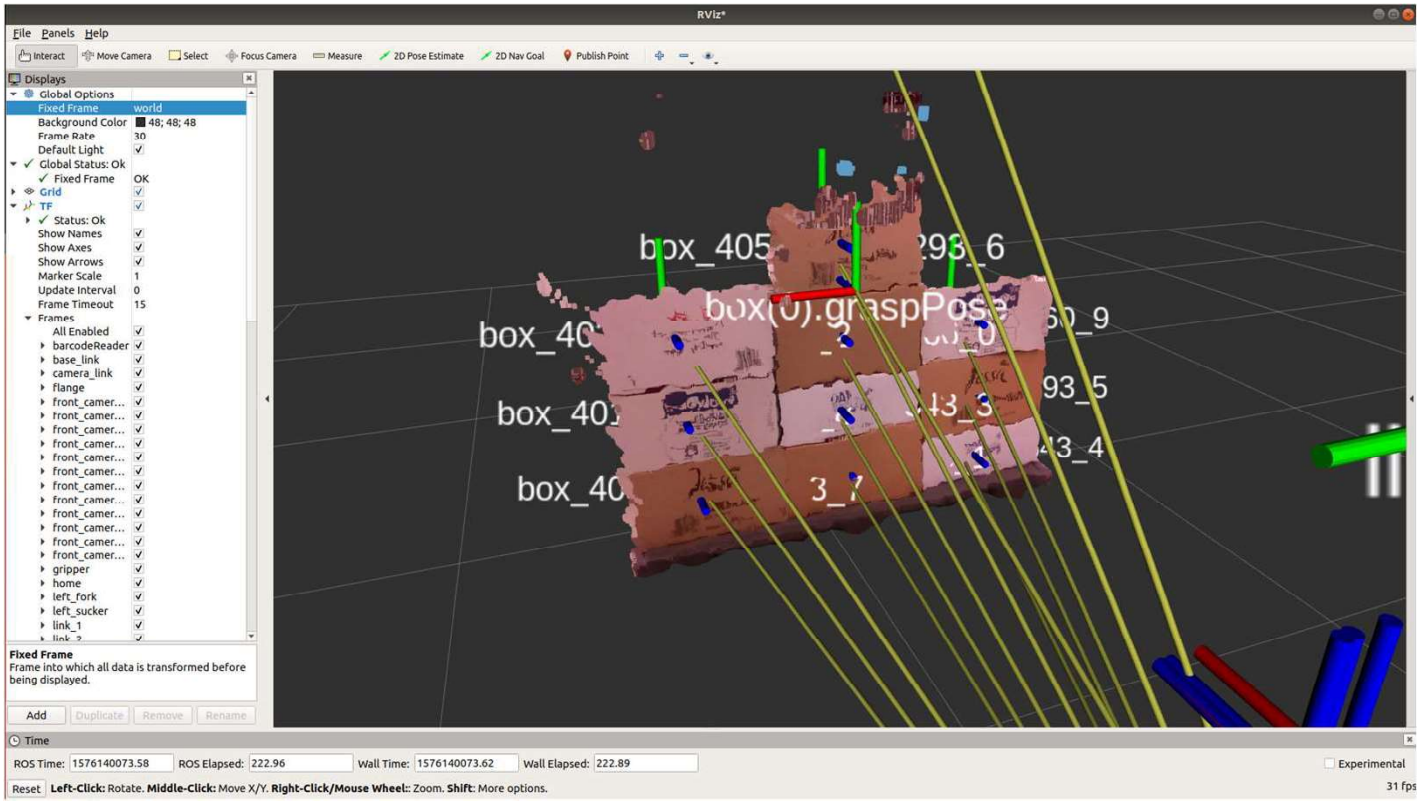
Turtlebot in rviz

```
rosrun rqt_graph rqt_graph
```

Introduction to ROS
ROS: rviz

Graphical visualization of topics.

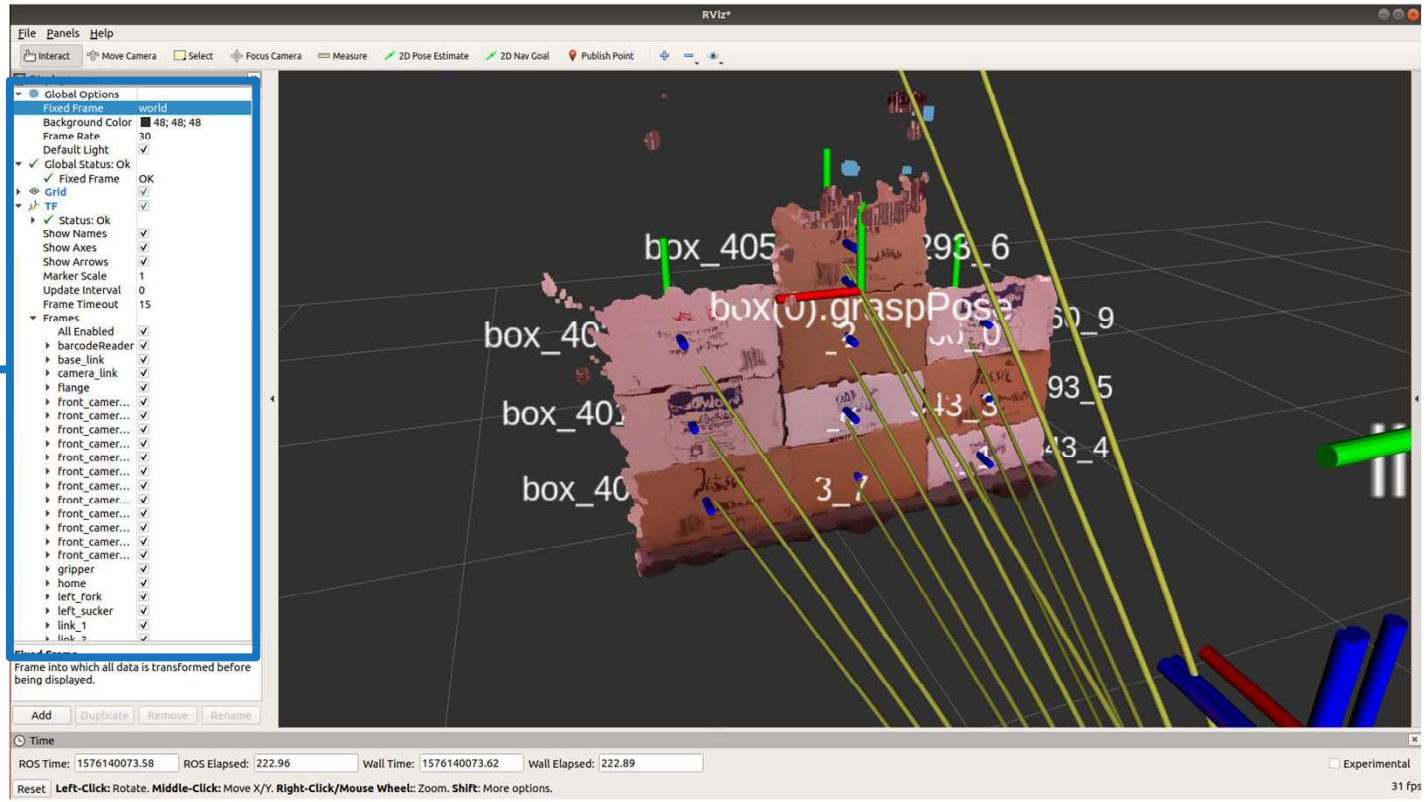
Rviz is a special node that subscribes to topics and provide to some of them a graphical visualization.



Introduction to ROS
ROS: rviz

Graphical visualization of topics.

List of visualized objects and topics

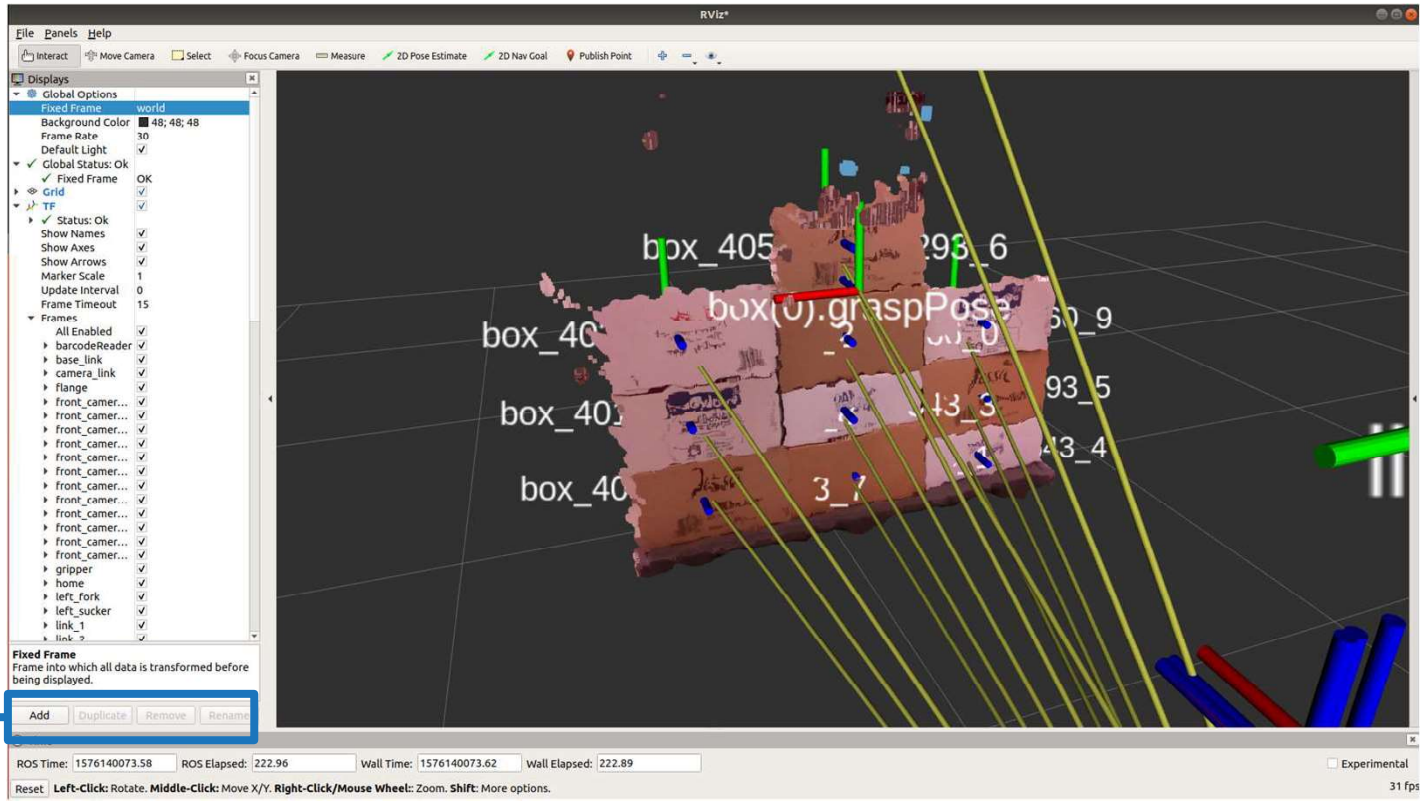


Introduction to ROS
ROS: rviz

Graphical visualization of topics.

List of visualized objects and topics

Add/remove topics from visualization



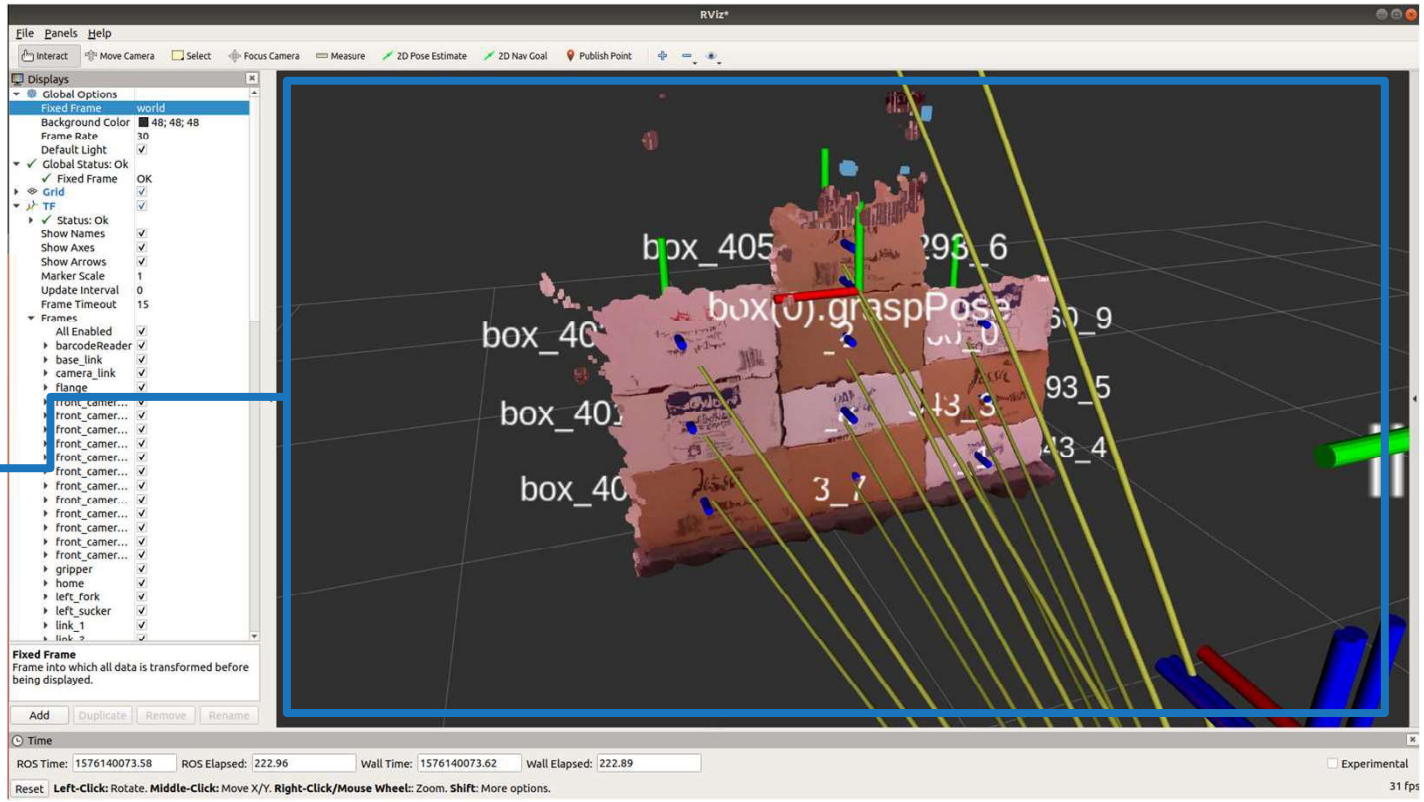
Introduction to ROS
ROS: rviz

Graphical visualization of topics.

List of visualized objects and topics

Add/remove topics from visualization

Visualization environment



Introduction to ROS
ROS: rviz

Graphical visualization of topics.

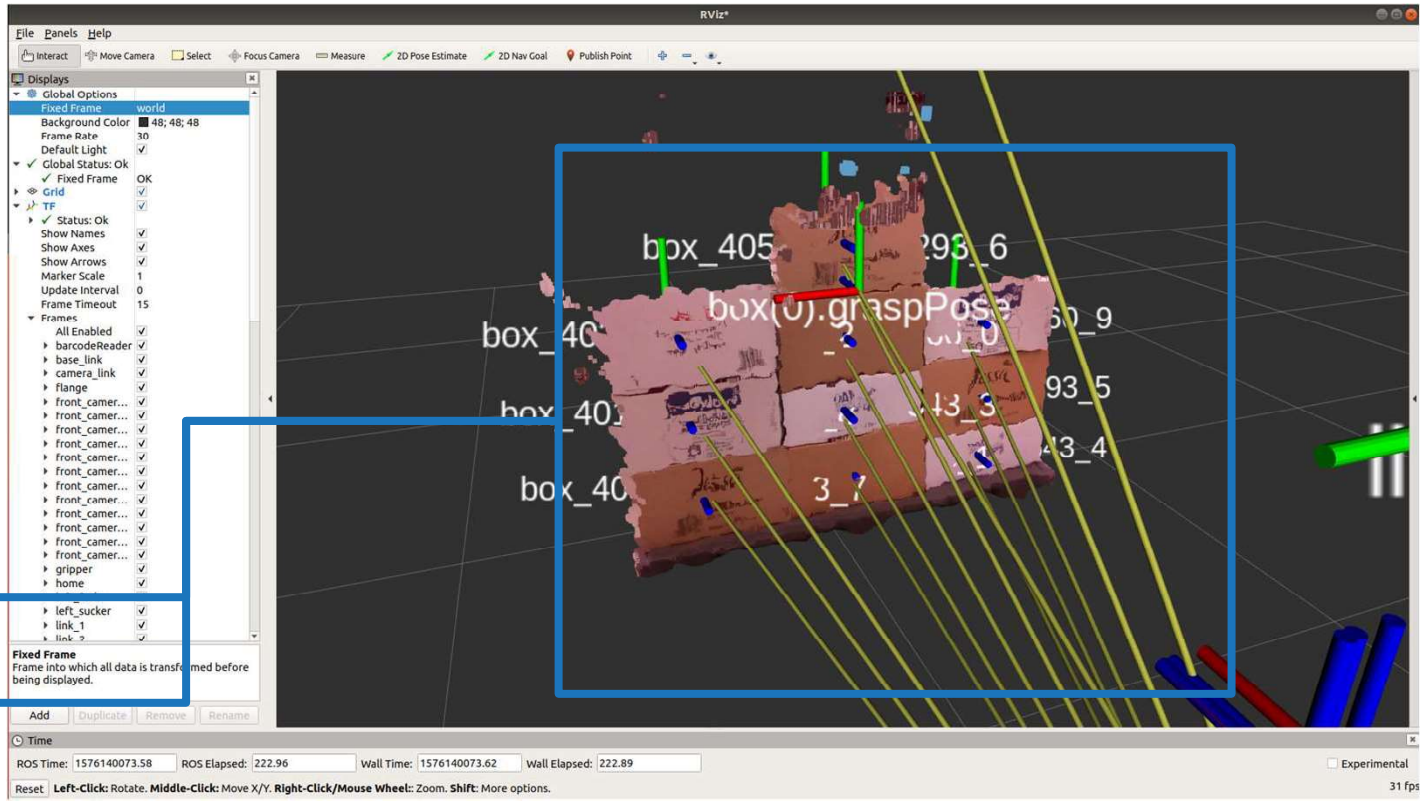
List of visualized objects and topics

Add/remove topics from visualization

Visualization environment

Colored pointcloud

Frames transforms



Introduction to ROS
ROS: Gmapping

http://wiki.ros.org/gmapping

```
sudo apt-get install ros-<version>-gmapping
```

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

```
roslaunch turtlebot_navigation gmapping_demo.launch
```

gmapping

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Turtlebot in teleoperation

```
roslaunch turtlebot_rviz_launchers view_robot.launch
```

Turtlebot in rviz, map, scan, odom

```
rosrun rqt_graph rqt_graph
```

Introduction to ROS
ROS: Gmapping

http://wiki.ros.org/gmapping

```
sudo apt-get install ros-<version>-gmapping
```

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

```
roslaunch turtlebot_navigation gmapping_demo.launch
```

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

gmapping

Turtlebot in teleoperation

Parameters:

map_update_interval (float, default: 5.0) – latency
delta (float, default: 0.05) – map resolution
maxUrange (float, default: 80.0) - sensor
iterations (int, default: 5) – scan matching
particles (int, default: 30) – particles
maxRange (float) – real sensor range
occ_thresh (float, default: 0.25) – threshold occupancy grid
resampleThreshold (float, default: 0.5)
…

Introduction to ROS
ROS: Map Server and Map Saver

http://wiki.ros.org/map_server

Maps are stored in a pair of files. The YAML file describes the map meta-data, and names the image file. The image file encodes the occupancy data.

The YAML format:

```
image: testmap.png
resolution: 0.1
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 0
```

Whether the white/black free/occupied semantics should be reversed

map_server is a ROS node that reads a map from disk and offers it via a ROS service.

```
rosrun map_server map_server mymap.yaml
```

```
rosrun map_server map_saver -f mymap
```

Map saver

Introduction to ROS
ROS: Map and Localization (AMCL)

http://wiki.ros.org/acml

MonteCarlo Localization

amcl is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach

```
sudo apt-get install ros-<version>-amcl
```

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

```
roslaunch turtlebot_gazebo amcl_demo.launch map_file:=<full path to map yaml file>
```

AMCL with map

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Turtlebot in teleoperation

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Turtlebot in rviz, map, scan, odom

Parameters:

min_particles (int, default: 100)
max_particles (int, default: 5000)
resample_interval (int, default: 2)
initial_pose_x (double, default: 0.0 meters), initial_cov_xx (double, default: 0.5*0.5 meters), …