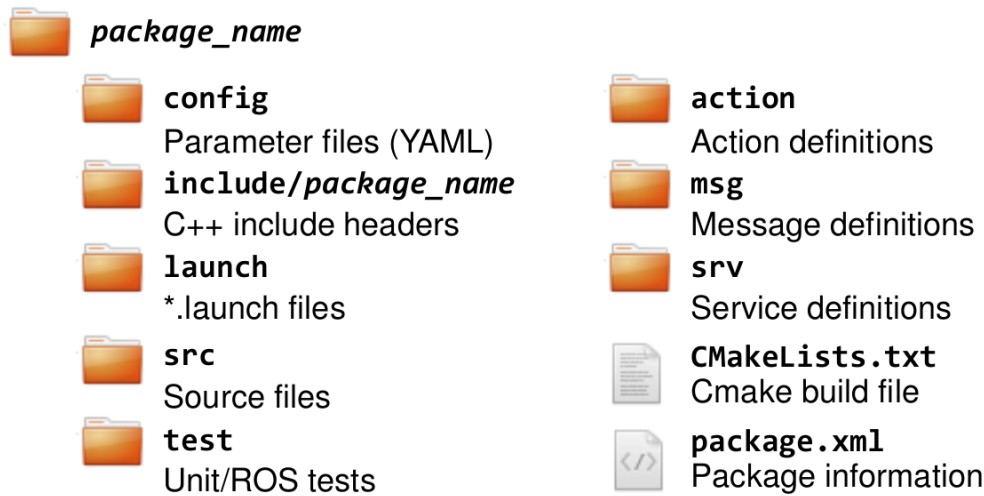


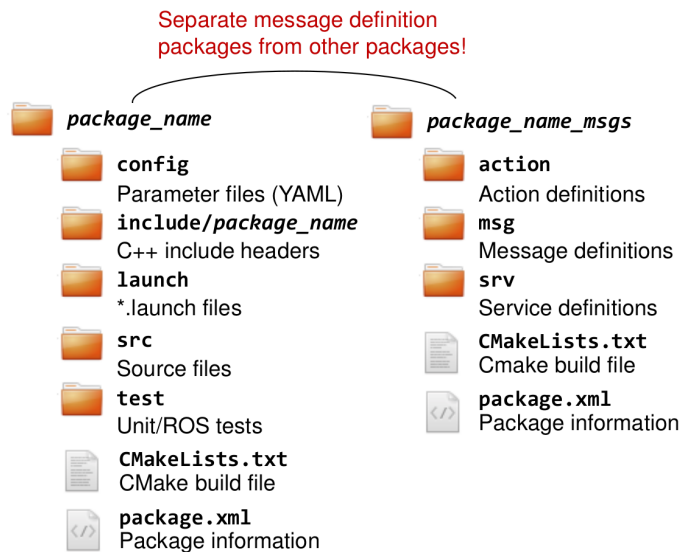
ROS Package folder



ROS Package folder

- Sometimes, in large projects, we may have a package only for messages, services and actions
- Why ?
 - It is not easy to tell which node owns a message.
 - It is easier when several people are working on the same project at different stages of progress.
 - The messages should change as little as possible so to not disturb the different developers. So this package can have stricter modification permissions. (Problem with ROS *bags*)

ROS Package folder



ROS: package.xml

- The **package.xml** file defines the **properties** of the package
 - Package name
 - Version number
 - Authors
 - Dependencies on other packages**
 - ...

```
<?xml version="1.0"?>
<package format="2">
  <name>ros_package_template</name>
  <version>0.1.0</version>
  <description>A ROS package</description>
  <maintainer email="pp@any...">Peter Paul</maintainer>
  <license>BSD</license>
  <url>https://github.com/toprobot/ros_...</url>
  <author email="pp@anybotics.com">Peter Paul</author>

  <buildtool_depend>catkin</buildtool_depend>

  <depend>roscpp</depend>
  <depend>sensor_msgs</depend>
</package>
```

ROS: CMakeLists.txt

- This file can be generated automatically by *catkin_create_pkg*.
- CMakeLists.txt : related to **CMake**, a **ROS-independent** tool for easily creating C++ MakeFile.
- Necessary even in **Python**, to declare the package, manage dependencies and installation.
- In a ROS project, they provide **special macros** for ROS/Catkin in Cmake : *add_messages_files()*, *catkin_package()*, etc.
- A **#** is used for **comments**.
- In a file generated with *catkin_create_pkg*, simply **uncomment** the commands you wish **to use**.
- In general, this file is not often modified during the development of a package.

ROS: CMakeLists.txt

- The CMakeLists.txt is the input to the CMakebuild system
- 1. Required CMake Version *cmake_minimum_required*
- 2. Package Name : *project()* → same as in *package.xml*
- 3. Find other CMake/Catkin packages needed for build : *find_package()* → list of libs, same as in *package.xml*
- 4. Message/Service/Action Generators to add your own stuff : *add_message_files()*, *add_service_files()*, *add_action_files()*
- 5. Invoke message/service/action generation : *generate_messages()* → list of msg dependencies
- 6. Allows the package to be used as a dependency in other projects : *catkin_package()*
- 7. Libraries/Executables to build : *add_library()* / *add_executable()* / *target_link_libraries()*
- 8. Tests to build : *catkin_add_gtest()*
- 9. Install rules : *install()*

Reference :
<http://wiki.ros.org/catkin/CMakeLists.txt>

```
cmake_minimum_required(VERSION 3.0.2)
project(ros_package_template)

## Use C++11
add_compile_options(-std=c++11)

## Find catkin macros and libraries
find_package(catkin REQUIRED COMPONENTS
  roscpp
  sensor_msgs
)

...

```

ROS: C++ CmakeLists.txt example

```
cmake_minimum_required(VERSION 3.0.2)
project(5ro14_husky_controller)
add_definitions(--std=c++11)

find_package(catkin REQUIRED COMPONENTS roscpp sensor_msgs)

catkin_package(
  INCLUDE_DIRS include
  # LIBRARIES
  CATKIN_DEPENDS roscpp sensor_msgs
  # DEPENDS
)

include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(${PROJECT_NAME}_node
src/MyNode.cpp src/MyController.cpp)

target_link_libraries(
${PROJECT_NAME}_node ${catkin_LIBRARIES})

```

- Use the same name as in the package.xml
- We use C++11 by default
- List the packages that your package requires to build (must be listed in package.xml)
- Specify build export information
 - INCLUDE_DIRS: Directories with exported header files
 - LIBRARIES: Exported libraries built in this project
 - CATKIN_DEPENDS: Other catkin projects that this project depends on
 - DEPENDS: Non-Catkin CMake projects that this project depends on (must be listed in package.xml)
- Specify locations of header files
- Declare a C++ executable
- Specify libraries to link the executable against

ROS: Python CmakeLists.txt example

```
cmake_minimum_required(VERSION 3.0.2)
project(5ro14_husky_controller_py)

find_package(catkin REQUIRED COMPONENTS rospy sensor_msgs)

catkin_package()

```

- Use the same name as in the package.xml
- List the packages that your package requires to build (must be listed in package.xml)
- catkin_package()* is required, but can be left empty as *rospy* does the rest.

Typical node : pseudo code v1

```
void callback_1(Msg1 msg) {... do stuff with msg from topic1...}

void callback_2(Msg2 msg) {... do stuff with msg from topic2...}

void main()
{
  ros::init("my_node");

  ros::Subscriber my_subscriber_1("topic1", callback_1);
  ros::Subscriber my_subscriber_2("topic2", callback_2);

  ros::Publisher my_publisher("topic3");

  while (ros::ok())
  {
    do_stuff();

    my_publisher.publish(my_msg);

    ros::spinOnce();
  }
}
```

**Warning: pseudo code !!
Do not use as is !**

31/01/2025

CSC_5RO14_TA - Emmanuel Battesti

13 / 37

Typical node : pseudo code v2

```
void callback_1(Msg1 msg) {... do stuff with msg from topic1 ...}

void callback_2(Msg2 msg) {... do stuff with msg from topic2
                             my_publisher.publish(other_msg);
                             ...}

void main()
{
  ros::init("my_node");

  ros::Subscriber my_subscriber_1("topic1", callback_1);
  ros::Subscriber my_subscriber_2("topic2", callback_2);

  ros::Publisher my_publisher("topic3");

  ros::spin(); ← Blocking function
}
```

**Warning: pseudo code !!
Do not use as is !**

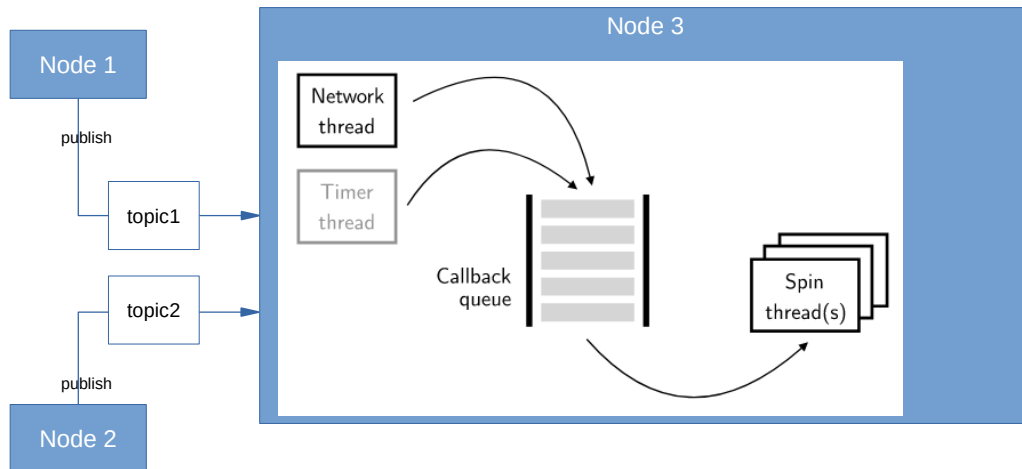
31/01/2025

CSC_5RO14_TA - Emmanuel Battesti

14 / 37

Callback queue

Callback : function to handle a message when it arrives



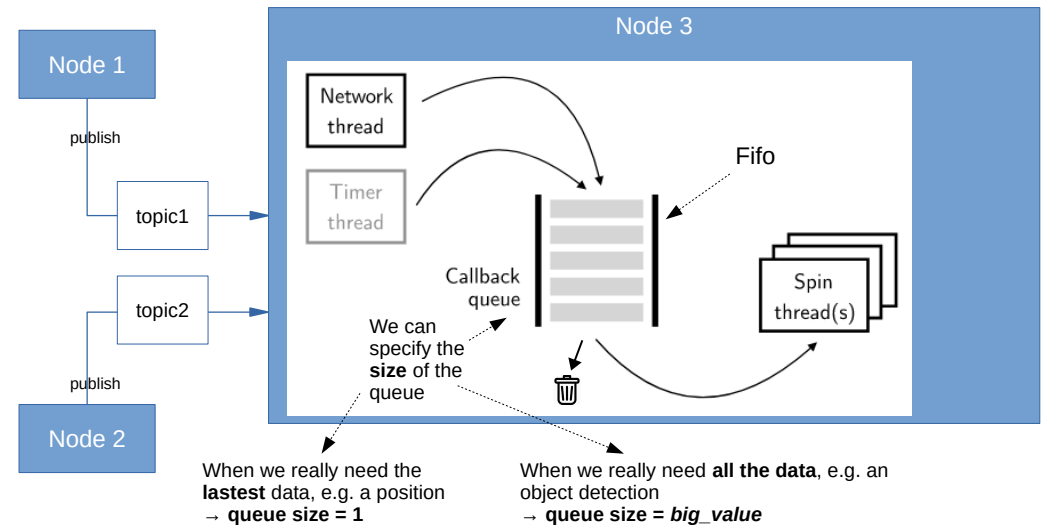
31/01/2025

CSC_5RO14_TA - Emmanuel Battesti

15 / 37

Callback queue

Callback : function to handle a message when it arrives



31/01/2025

CSC_5RO14_TA - Emmanuel Battesti

16 / 37

ROS Callback Handling

- Single-Threaded Spinner (**default**):
 - Each topic has its own callback queue.
 - Callbacks are processed sequentially in the order messages arrive for each topic.
 - Issue: A long callback can delay the processing of other topics.
- Multi-Threaded Spinner:
 - Allows parallel callback processing.
 - Multiple threads handle callbacks, preventing delays caused by long-running callbacks in other topics.

ROS C++ Client Library (roscpp) : code example

```
#include <ros/ros.h>

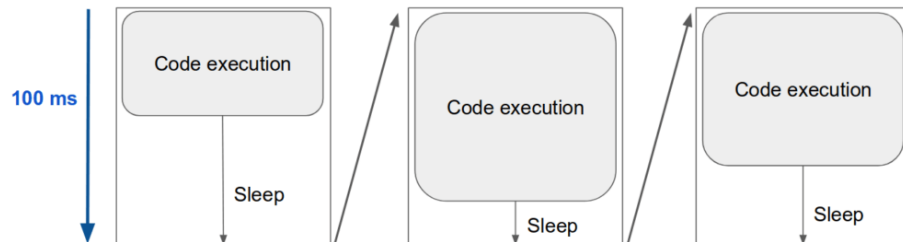
int main(int argc, char** argv)
{
    ros::init(argc, argv,
"hello_world");
    ros::NodeHandle nodeHandle;
    ros::Rate loopRate(10);

    unsigned int count = 0;
    while (ros::ok()) {
        ROS_INFO_STREAM("Hello
World " << count);
        ros::spinOnce();
        loopRate.sleep();
        count++;
    }

    return 0;
}
```

- ROS main **header** file include
- **ros::init(...)** must be called before calling any other ROS functions.
- The node **handle** is the entry point for communication with the ROS system (topics, services, parameters),
- **ros::Rate** is a helper class to run loops at a desired frequency,
- **ros::ok()** checks if a node should continue running
Returns false if SIGINT is received (Ctrl + C) or **ros::shutdown()** is called
- **ROS_INFO()** **logs** messages to the file system
- **ros::spinOnce()** processes incoming messages via callbacks

Roscpp - ros::Rate



Roscpp - ros::spin() vs ros::spinOnce()

- **ros::spinOnce()** calls the callbacks waiting to be called at that point in time.
- **ros::spin()** gives control over to ROS, which allows it to call user callbacks.
- It is a **blocking** function : it will not return until the node is shut down, by calling to **ros::shutdown()** or by pressing Ctrl-C.
- Internally, **ros::spin()** looks like :

```
ros::Rate loopRate(10);
while (ros::ok()) {
    ros::spinOnce();
    loopRate.sleep();
    count++;
}
```

Roscpp - Subscriber

- Start listening to a topic by calling the method `subscribe()` of the node handle

```
ros::Subscriber subscriber =  
nodeHandle.subscribe(topic,  
queue_size, callback_function);
```

- When a message is received, a callback function is called with the content of the message as an argument
- `ros::spin()` processes callbacks and will not return until the node has been shut down

listener.cpp

```
#include "ros/ros.h"  
#include "std_msgs/String.h"  
  
void chatterCallback(const std_msgs::String&  
msg)  
{  
    ROS_INFO("I heard: [%s]", msg.data.c_str());  
}  
  
int main(int argc, char **argv)  
{  
    ros::init(argc, argv, "listener");  
    ros::NodeHandle nodeHandle;  
  
    ros::Subscriber subscriber =  
nodeHandle.subscribe("chatter", 10,  
chatterCallback);  
    ros::spin();  
    return 0;  
}
```

31/01/2025

CSC_5RO14_TA - Emmanuel Battesti

21 / 37

Roscpp - Publisher

- Create a publisher using the node handle

```
ros::Publisher publisher =  
nodeHandle.advertise<message_type>  
(topic, queue_size);
```

- Create the message content
- Publish the content with

```
publisher.publish(message);
```

Reference :
https://github.com/ros/ros_tutorials/blob/melodic-devel/roscpp_tutorials/talker/talker.cpp

talker.cpp

```
#include <ros/ros.h>  
#include <std_msgs/String.h>  
  
int main(int argc, char **argv) {  
    ros::init(argc, argv, "talker");  
    ros::NodeHandle nh;  
    ros::Publisher chatterPublisher =  
nh.advertise<std_msgs::String>("chatter", 1);  
    ros::Rate loopRate(10);  
  
    unsigned int count = 0;  
    while (ros::ok()) {  
        std_msgs::String message;  
        message.data = "hello world " +  
std::to_string(count);  
        ROS_INFO_STREAM(message.data);  
        chatterPublisher.publish(message);  
        ros::spinOnce();  
        loopRate.sleep();  
        count++;  
    }  
    return 0;  
}
```

31/01/2025

CSC_5RO14

ROS C++ Client Library : Object Oriented Programming

MyNode.cpp

```
#include <ros/ros.h>  
#include "my_package/MyWork.hpp"  
  
int main(int argc, char** argv)  
{  
    ros::init(argc, argv, "my_node_name");  
    ros::NodeHandle nodeHandle("~");  
  
    my_package::MyWork myWork(nodeHandle);  
  
    ros::spin();  
    return 0;  
}
```

Specify a function handler to a method from within the class (here **MyWork**) as:

```
subscriber_ = nodeHandle_.subscribe(topic, queue_size, &MyWork::one_callback, this);
```

- We can have those files :
MyNode.cpp, MyWork.cpp, MyWork.hpp, Algorithm.cpp, Algorithm.hpp
- class **MyWork** : Main node class providing the ROS interface (subscribers, parameters, timers etc.)
- class **Algorithm** : Class implementing the algorithmic part of the node. Note: The algorithmic part of the code could be separated in a (ROS-independent) library

31/01/2025

CSC_5RO14_TA - Emmanuel Battesti

23 / 37

Graph resource names 1/2

- Nodes, topics, services, and parameters = **graph resources**. Their name = **graph resource name**
- Namespaces** are used to group related graph resources together. A **base name** describes the resource itself. e.g.
`/namespace1/namespace2/baseName`
- Global Names** : starting with a « / ».
 - For example :
 - `/turtle1/cmd_vel`
 - `/teleop_turtle`
- Relative names** : **not** starting with a « / ». Not fully defined, the name should be resolved.
 - Relative names make it easier to build complicated systems by composing smaller parts.
 - Default namespace + relative name = global name
 - e.g. `./turtle1` + `cmd_vel` = `/turtle1/cmd_vel`
 - e.g. `./turtle1` + `abc/cmd_vel` = `/turtle1/abc/cmd_vel`

31/01/2025

CSC_5RO14_TA - Emmanuel Battesti

24 / 37

Graph resource names 2/2

- **Private names** : starting with a ~. Not fully defined, the name should be resolved.
 - Like *relative names* but use the **name of their node** as a namespace
 - Node name + ~private_name = global name
 - e.g. : /sim1/pubvel + ~max_vel = /sim1/pubvel/max_vel
 - *Private names* are often used **for parameters**

Roscpp - Node Handle

- Acts as the **interface between your program and the ROS** system.
- Communication:
 - Topic : Create publishers and subscribers.
 - Service : Advertise and handle services.
- Access Parameters from the Parameter Server.
- Usage Example:
 - Publisher: ros::Publisher pub = **nh**.advertise<std_msgs::String>("topic", 1000);
 - Subscriber: ros::Subscriber sub = **nh**.subscribe("topic", 1000, callback);

Roscpp - Node Handle

- Different types of node handles
 1. Default (public) node handle:


```
nh_ = ros::NodeHandle();
```
 2. Private node handle:


```
nh_private_ = ros::NodeHandle("~");
```
 3. Namespaced node handle:


```
nh_foo_ = ros::NodeHandle("foo");
```
 4. Private node handle:


```
nh_privfoo_ = ros::NodeHandle("~foo");
```
 5. Global node handle:


```
nh_global_ = ros::NodeHandle("/");
```
- For a **node** in a namespace **ns** looking up **topic**, these will resolve to:
 - ⇒ /ns/topic
 - ⇒ /ns/node/topic
 - ⇒ /ns/foo/topic
 - ⇒ /ns/node/foo/topic
 - ⇒ /topic

Roscpp - Logging

- Mechanism for logging human readable text from nodes in the console and to log files
- Instead of std::cout, use e.g. ROS_INFO_STREAM
- Automatic logging to console, log file, and /rosout topic
- Different severity levels (Info, Warn, Error etc.)
- Supports both printf-style and stream-style formatting


```
ROS_INFO("Result: %d", result);
ROS_INFO_STREAM("Result: " << result);
```
- Other features like conditional, throttled, delayed logging etc.

ROS Parameter Server

- Nodes use the *parameter server* to **store** and **retrieve** parameters at runtime
- Best used for **static data** such as configuration parameters
- Parameters can be defined in **launch files** or separate **YAML files**
- Launch file can **load** YAML files

config.yaml

```
camera:
  left:
    name: left_camera
    exposure: 1.0
  right:
    name: right_camera
    exposure: 1.1
```

package.launch

```
<launch>
  <node name="name" pkg="package" type="node_type">
    <rosparam command="load" file="$(find package)/config/config.yaml" />
    <param name="camera/left/exposure" type="double" value="2.0" />
    <rosparam param="camera/left/exposure">3.0</rosparam>
    <rosparam>
      camera/left/exposure: 4.0
    </rosparam>
  </node>
</launch>
```

31/01/2025

CSC_5RO14_TA - Emmanuel Battesti

29 / 37

ROS Parameter Server

- List all parameters with
`> rosparam list`
- Get the value of a parameter with
`> rosparam get parameter_name`
- Set the value of a parameter with
`> rosparam set parameter_name value`

config.yaml

```
camera:
  left:
    name: left_camera
    exposure: 1.0
  right:
    name: right_camera
    exposure: 1.1
```

- For example, if *config.yaml* has been loaded, to get the exposure from the left camera:

```
> rosparam get camera/left/exposure
```

31/01/2025

CSC_5RO14_TA - Emmanuel Battesti

30 / 37

ROS Parameter Server : C++ API

- Get a parameter in C++ with
`nodeHandle.getParam(parameter_name, variable)`
- Method returns `true` if parameter was found, `false` otherwise
- Global and relative parameter access:
 - Global parameter name with preceding /
`nodeHandle.getParam("/camera/left/exposure", variable)`
 - Relative parameter name (relative to the node handle)
`nodeHandle.getParam("camera/left/exposure", variable)`
- For parameters, typically use the private node handle :
`ros::NodeHandle("~")`

```
ros::NodeHandle nodeHandle("~");
std::string myParam;
If (!nodeHandle.getParam("myParam", myParam)) {
  ROS_ERROR("Could not find myParam parameter!");
}
```

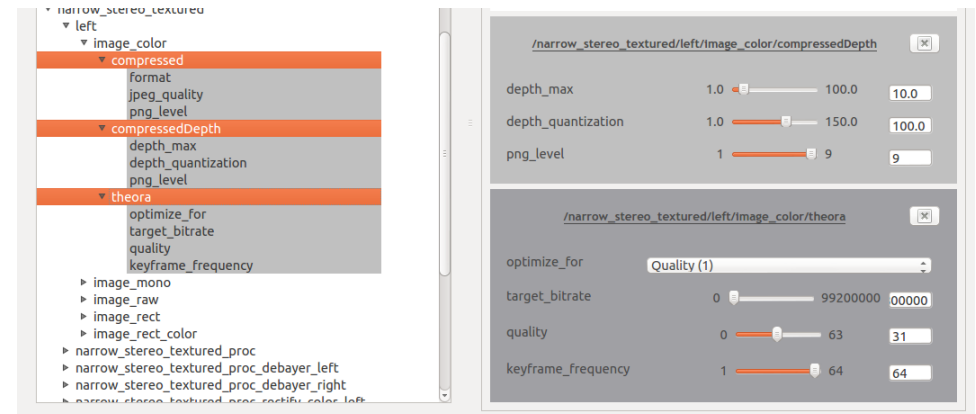
31/01/2025

CSC_5RO14_TA - Emmanuel Battesti

31 / 37

ROS Parameter : dynamic reconfigure

- The package `dynamic_reconfigure` permit to use parameter dynamically.



31/01/2025

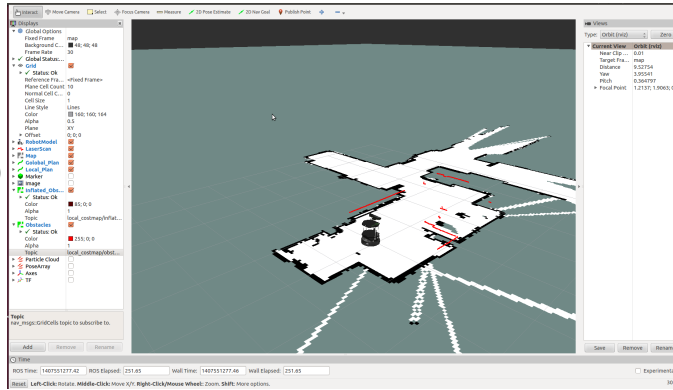
CSC_5RO14_TA - Emmanuel Battesti

32 / 37

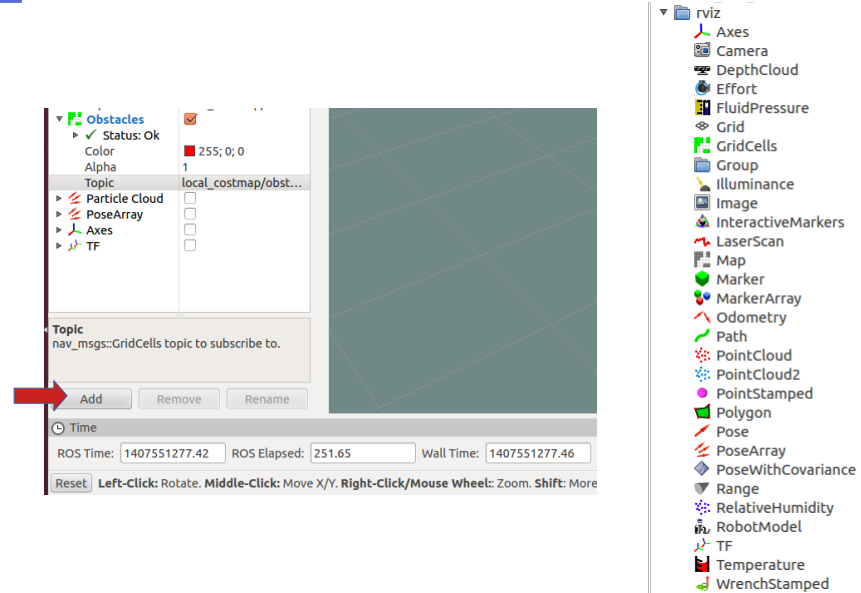
RViz

- 3D visualization tool for ROS
- Subscribes to topics and visualizes the message contents
- Different camera views (orthographic, topdown, etc.)
- Interactive tools to publish user information
- Save and load setup as RViz configuration
- Extensible with plugins
- Run RViz with

```
> rosrn rviz rviz
```



Rviz : Display Plugins

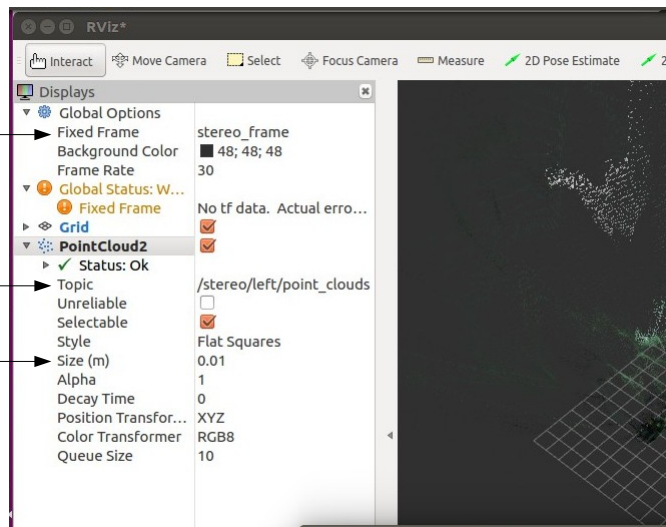


Rviz :Visualizing Point Clouds Example

Frame in which the data is displayed (has to exist!)

Choose the topic for the display

Change the display options (e.g. size)



Exercice 1 – Playing with husky (part 2)

- Topics covered :
 - ROS package structure
 - Integration and programming
 - ROS C++ client library (roscpp)
 - ROS subscribers and publishers
 - ROS parameter server
 - RViz visualization

Further References

- **ROS Wiki:**
 - <http://wiki.ros.org/>
- **Installation:**
 - <http://wiki.ros.org/ROS/Installation>
- **Tutorials:**
 - <http://wiki.ros.org/ROS/Tutorials>
- **Available packages:**
 - <http://www.ros.org/browse/>
- **ROS Cheat Sheet :**
 - <https://www.clearpathrobotics.com/ros-robot-operating-system-cheat-sheet/>
 - https://kapeli.com/cheat_sheets/ROS.docset/Contents/Resources/Documents/index
- **ROS Best Practices :**
 - https://github.com/leggedrobotics/ros_best_practices/wiki
- **ROS Package Template :**
 - https://github.com/leggedrobotics/ros_best_practices/tree/master/ros_package_template