ROS Notes - SOFAR Lectures

Davide Lanza

June 12, 2019

Contents

Chapter 1

Introduction

1.1 What is ROS

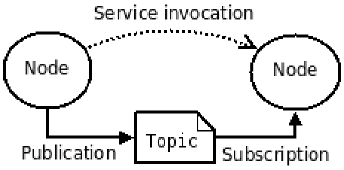
The **R**obot **O**perating **S**ystem is a **middleware** software: it provides services to applications beyond those of the operating system.

A ROS application is made of a number of processes, which can run on several hosts, are connected at run-time peer-to-peer and there is no central server. So, there exists a central entity: the **ROS master** that connects the processes at **startup and** also **connects** processes each time a new process is created, **but later**, communication is **peer-to-peer**.

1.2 Nodes, Topics and Messages

1.2.1 Basic concepts

Nodes: basic modules.
ROSMaster: core process, connects modules.
Messages: the way modules communicate.
Topics: communication channels for messages.
Parameters: node customization tools, global parameters, etc...
Services: computations on request.



1.2.2 Nodes

A ROS node is an **executable** program with a well-defined purpose which uses the ROS framework for execution. It is **modular**, so it is individually compiled and executed.

Nodes can execute on **separate machines**, transparently to the programmer.

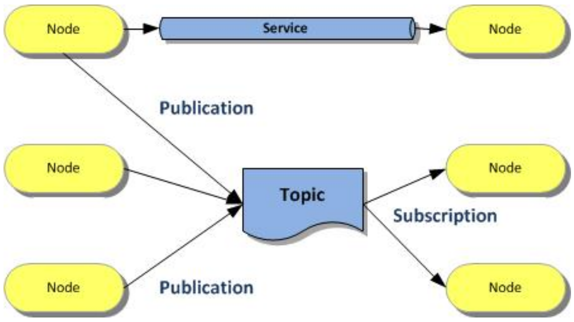
At creation, nodes are **connected** to other existing nodes by the **`rosmaster`** process. `rosmaster` is a kind of name server. It has to be running to run a node. Rosmaster runs in `localhost` by default.

Nodes are **written** with the help of a **ROS client library**:
→ libraries let you write ROS nodes, publish to topics, subscribe to topics, write services, call

services, use the parameter server etc... (see later for the explanation of these terms). The main clients are `roscpp` and `rospy`.

1.2.3 Subscription and publication

A node may **subscribe** to (i.e. listen to) any number of topics (the corresponding topics are the **inputs** to the node) and **publish** to (i.e. send messages to) any number of topics (the corresponding topics are the **outputs** of the node.



A node may subscribe to 0 to N topics and may publish to 0 to N topics.

1.2.4 Node tasks

A node may perform different tasks, like interfacing to a **sensor** and publish its raw data, **control** an actuator or execute any kind of specific **algorithm** (planning, calculate a robot model, process raw sensor data to extract information...). Nodes can also implement some visualization tool and **simulate** the dynamics of a system.

1.2.5 Topics

Nodes communicate by sending messages to topics and listening to topics. **A topic defines a type of message**.

It's a 1-to-N communication: the information published by one node to a topic is available to all subscribers to the topic. A node who subscribes to a topic receives all the information published to that topic (possibly by several publishers).

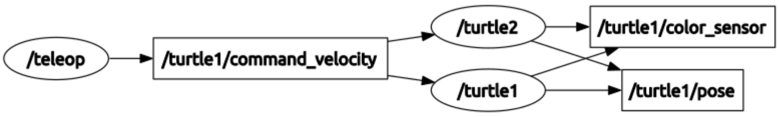
A **message** is a **strictly typed data structure**. There are many pre-defined message types for common data (Pose, quaternion, transformation... Twist... Point clouds, images...) but new types of messages can be defined if necessary. [1]

1.2.6 Connection graph

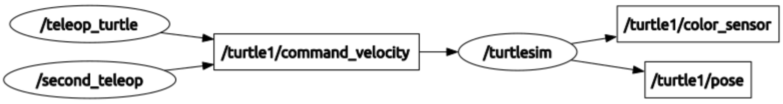
The **`rqt_graph`** tool allows checking the connections between nodes via topics. In the following Figure, the **ellipses** are the **nodes** and the **rectangles** the **topics**:

---

[1] As much as possible, use pre-defined message types.



One publisher of "command_velocity", two subscribers



Two publishers of "command_velocity", one subscriber

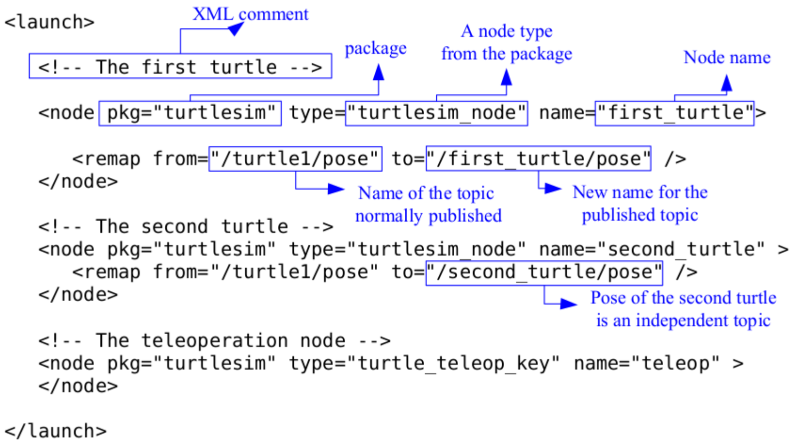
This connection graph is a fundamental tool in the development of a ROS application. Your first goal is to obtain the proper graph (i.e. the proper connections between nodes).[2]

Learning how to use `rqt_graph` tool is very important since the graph of complex applications is usually big.

1.3 ROS Files

1.3.1 Launch files

Launching each node in a different window or thumbnail is **long** and the process is **error**-prone. Moreover, the screen becomes messy with too many windows. **Launch files** (written in XML) are the solution to this problem:



```
<launch>

   <!-- The first turtle -->

   <node pkg="turtlesim" type="turtlesim_node" name="first_turtle">

      <remap from="/turtle1/pose" to="/first_turtle/pose" />
   </node>

   <!-- The second turtle -->
   <node pkg="turtlesim" type="turtlesim_node" name="second_turtle" >
      <remap from="/turtle1/pose" to="/second_turtle/pose" />
   </node>

   <!-- The teleoperation node -->
   <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" >
   </node>

</launch>
```

It is notable a field called `remap`. It means **remapping** and here is how to interpret [3] the `from` and `to` fields:

- *For a topic that the node publishes*
  `from` should contain what the published topic name would be **without any remapping**
  `to` should be the name you actually want to be visible in the network (listed by `rostopic list` command).

- *For a topic that the node subscribes to*
  `from` should contain the name that would be expected by the node should **no remapping** be performed
  `to` should be a topic name actually present in the network.

---

[2] For an even moderately complex application, it is a good idea to first draw the graph you mean to obtain.

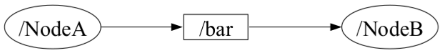
[3] Many students are initially confused by remappings. Try to keep these rules in mind. Re-read this part when necessary in the early stages of the labs.

Example Let's consider this situation originally without remapping:

1) Now, let's consider this first `.launch` file:

```
<node pkg="packageA" type="packageA_node" name="nodeA">
    <remap from="/foo" to="/bar" />
</node>
<node pkg="packageB" type="packageB_node" name="nodeB">
</node>
```

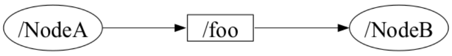
We will obtain:



2) Now, let's consider this other `.launch` file:

```
<node pkg="packageA" type="packageA_node" name="nodeA">
</node>
<node pkg="packageB" type="packageB_node" name="nodeB">
    <remap from="/bar" to="/foo" />
</node>
```

We will obtain:



3) Finally, let's consider this `.launch` file:

```
<node pkg="packageA" type="packageA_node" name="nodeA">
    <remap from="/foo" to="/foobar" />
</node>
<node pkg="packageB" type="packageB_node" name="nodeB">
    <remap from="/bar" to="/foobar" />
</node>
```

We will obtain:

As illustrated by these three examples, there are always multiple solutions to a remapping problem. Obtaining **clear topic names** is more important than minimizing the number of remappings.

There are ways for the programmer to ease the work of the package/node users by a careful naming of the topics because:

- Node names may be automatically prepended to the topic name (`/nodeName/topicName`).
- Topic names can be made sensitive to the namespace in which the node is run.

With the launch file there is **no need** to explicitly launch the ROS master with **`roslaunch`**, because either `roslaunch` uses an already running one if any (case of the Baxter) or `roslaunch` launches a `roscore` otherwise.

The **node name cannot** specify a **namespace** (~~`/robot/something`~~) and **cannot** have an initial "/".

Anything which takes place between `<node>` and `</node>` is **local** to the node (e.g. name remapping).

Use **`rqt_graph`** to check the connections between nodes. If nodes you wanted to be connected are not, then some remapping remains to be done.

1.3.2 Messages

As we already saw, essages are strictly typed data structures for inter-node communication, that can include primitive types (boolean, integer, floating-point), arrays of primitive types, nested structures and arrays, like in C.

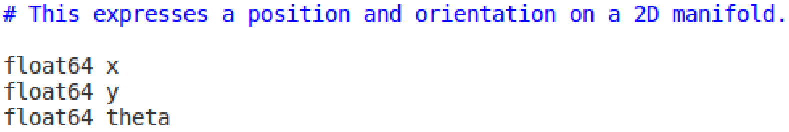
ROS predefines may useful types of messages, but programmers can define their own messages whenever a predefined message is not available.

Examples Let's consider these different messages:

1) Pose2D Message :

`File: geometry_msgs/Pose2D.msg`

The **raw** message definition is:



```
# This expresses a position and orientation on a 2D manifold.

float64 x
float64 y
float64 theta
```

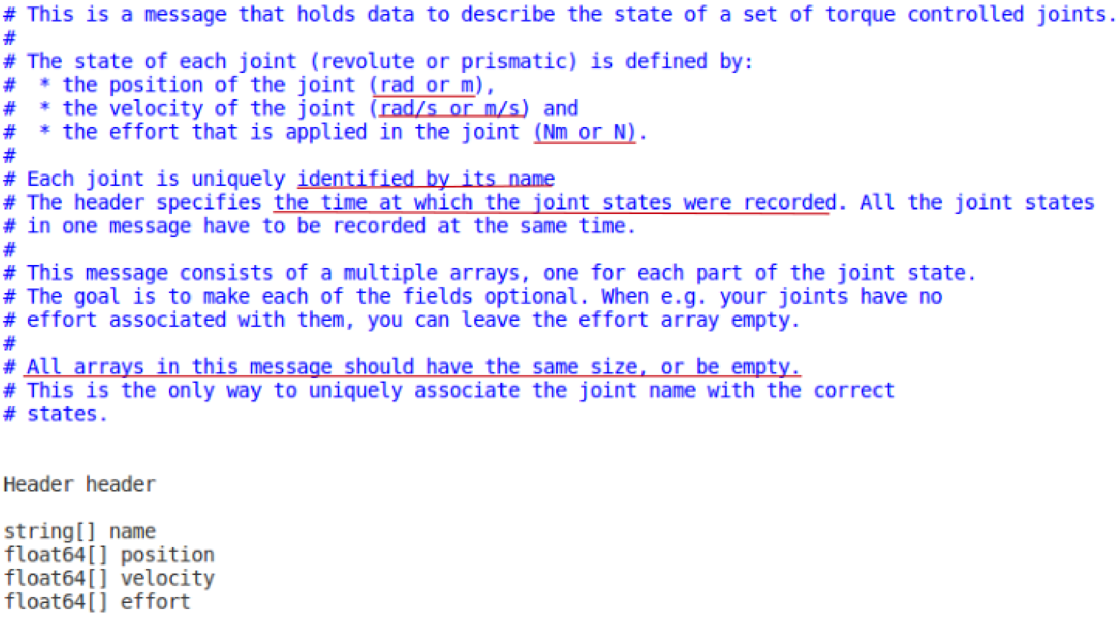
We can have also a **compact** definition:

```
float64 x
float64 y
float64 theta
```

2) JointState Message :

`File: geometry_msgs/JointState.msg`

The **raw** message definition is:



```
# This is a message that holds data to describe the state of a set of torque controlled joints.
#
# The state of each joint (revolute or prismatic) is defined by:
#  * the position of the joint (rad or m),
#  * the velocity of the joint (rad/s or m/s) and
#  * the effort that is applied in the joint (Nm or N).
#
# Each joint is uniquely identified by its name
# The header specifies the time at which the joint states were recorded. All the joint states
# in one message have to be recorded at the same time.
#
# This message consists of a multiple arrays, one for each part of the joint state.
# The goal is to make each of the fields optional. When e.g. your joints have no
# effort associated with them, you can leave the effort array empty.
#
# All arrays in this message should have the same size, or be empty.
# This is the only way to uniquely associate the joint name with the correct
# states.


Header header

string[] name
float64[] position
float64[] velocity
float64[] effort
```

For the **Baxter**, for example, a sample joint state could be:

```
---
header:
   seq: 609991
   stamp:
      secs: 1394012854
      nsecs: 243818360
   frame_id: ''
name:   ['head_nod', 'head_pan',
          'left_e0', 'left_e1',
          'left_s0', 'left_s1',
          'left_w0', 'left_w1', 'left_w2',
          'right_e0', 'right_e1',
          'right_s0', 'right_s1',
          'right_w0', 'right_w1', 'right_w2',
          'torso_t0']
Position:  [0.0, ... suppressed ...]
Velocity:  [0.0, -0.0179763373374939,... suppressed ...]
Effort:    [0.0, 0.0, -9.556, ... suppressed ...]
---
```

1.4 Command line tools

Here a list of the main ROS commands for **nodes**:

- `rosnode list` : list all nodes
- `rosnode ping <node>` : test whether a node is reachable
- `rosnode info <node>` : print info about a node
- `rosnode machine` : list machines in the configuration
- `rosnode machine <machine_name>` : list nodes running on a given machine
- `rosnode kill <node>` : kill a running node
- `rosnode cleanup` : remove unreachable nodes from registration information
- `rosnode <command> -h` : get command help

Here a list of the main ROS commands for **topics**:

- `rostopic list` : list all active topics
- `rostopic info <topic>` : print info about a topic
- `rostopic type <topic>` : display topic type
- `rostopic hz <topic>` : print publishing rate
- `rostopic echo <topic>` : print topic contents to screen
- `rostopic bw <topic>` : print bandwidth used by a topic
- `rostopic <topic> -h` : print topic specific help

Chapter 2

Workspace initialization

2.1 Create a ROS Workspace

Let's create and build a catkin workspace:

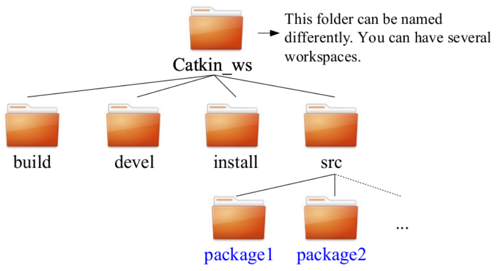
```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

In our case:

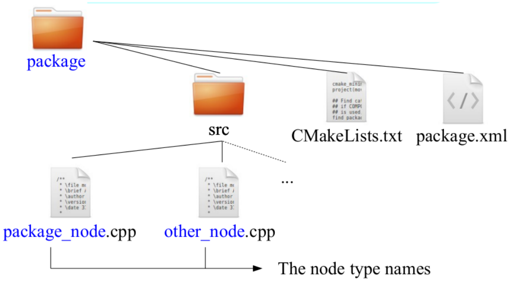
```
$ mkdir -p workspace/sofar_catkin_ws/src
$ cd workspace/sofar_catkin_ws/
$ catkin_make
```

The **catkin_make** command is a convenience tool for working with catkin workspaces. Running it the first time in your workspace, it will create a `CMakeLists.txt` link in your `'src'` folder. Additionally, if you look in your current directory you should now have a `'build'` and `'devel'` folder.

Inside the `'devel'` folder you can see that there are now several `setup.*sh` files. Sourcing any of these files will overlay this workspace on top of your environment. To understand more about this see the general catkin documentation at www.wiki.ros.org/catkin.



The names of the folders in the catkin_ws/src folder are the package names



Example: <node pkg="package" type="other_node" name="foo"> </node>

Before continuing source your new `setup.*sh` file:

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

To make sure your workspace is properly overlayed by the setup script, make sure `ROS_PACKAGE_PATH` environment variable includes the directory you're in with the following command:

```
$ echo $ROS_PACKAGE_PATH
```

We should obtain

```
/home/youruser/catkin_ws/src:/opt/ros/kinetic/share
```

In our case we obtained:

```
/home/davidelanz/workspace/sofar_catkin_ws/src:/opt/ros/melodic/share
```

Error "setup.bash not found": it can happen to have an error like:

```
bash: /home/user/catkin_ws/devel/setup.bash: No such file or directory
```

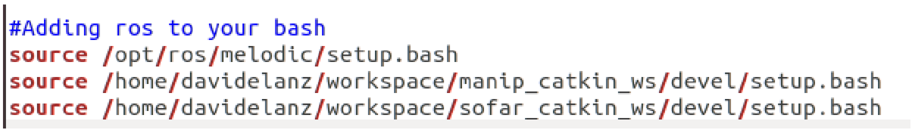
In this case the program has left some unnecessary lines in the `/.bashrc` file. This file `/home/user/catkin_ws/devel/setup.bash` could have been added by the command like:

```
$ echo "source/opt/ros/jade/setup.bash" >> ~/.bashrc.
```

Use this command to find and delete them:

```
gedit ~/.bashrc
```

In here we can add and remove the setup files to add to the bash:



```
#Adding ros to your bash
source /opt/ros/melodic/setup.bash
source /home/davidelanz/workspace/manip_catkin_ws/devel/setup.bash
source /home/davidelanz/workspace/sofar_catkin_ws/devel/setup.bash
```

2.2 Navigating the ROS Filesystem

To inspect a package we need ros-tutorials (`<distro>` for us is `melodic`):

```
$ sudo apt-get install ros-<distro>-ros-tutorials
```

Remind:
– *Packages*: Packages are the software organization unit of ROS code. Each package can contain libraries, executables, scripts, or other artifacts.
– *Manifests* (`package.xml`): A manifest is a description of a package. It serves to define dependencies between packages and to capture meta information about the package like version, maintainer, license, etc...

Code is spread across many ROS packages. Navigating with command-line tools such as ls and cd can be very tedious which is why ROS provides you **Filesystem Tools** to help you.

○ **`rospack`** allows you to get information about packages ($ is the command, without is the console message):

```
$ rospack find [package_name]
YOUR_INSTALL_PATH/share/roscpp
```

An example:

```
$ rospack find roscpp
/opt/ros/melodic/share/roscpp
```

○ **`roscd`** is part of the **`rosbash`** suite. It allows you to change directory (**`cd`**) directly to a package or a stack:

```
$ roscd [locationname[/subdir]]
```

To verify that we have changed to the `roscpp` package directory, run this example:

```
$ roscd roscpp
```

Now let's print the working directory using the Unix command `pwd`[1]:

```
$ pwd
YOUR_INSTALL_PATH/share/roscpp
```

`roscd` can also move to a **subdirectory** of a package or stack:

```
$ roscd roscpp/cmake
$ pwd
YOUR_INSTALL_PATH/share/roscpp/cmake
```

○ **echo and ROS_PACKAGE_PATH** from the previous example you can see that `YOUR_INSTALL_PATH/share/roscpp` is the same path that `rospack find` gave in the corresponding example.

Note that `roscd`, like other ROS tools, will only find ROS packages that are within the directories listed in your `ROS_PACKAGE_PATH`. To see what is in your `ROS_PACKAGE_PATH`, type:

```
$ echo $ROS_PACKAGE_PATH
```

Your `ROS_PACKAGE_PATH` should contain a list of directories where you have ROS packages separated by colons. Similarly to other **environment paths**, you can add additional directories to your ROS_PACKAGE_PATH, with each path separated by a colon `':'`.

○ **roscd log** will take you to the folder where ROS stores log files. Note that if you have not run any ROS programs yet, this will yield an error saying that it does not yet exist.

○ **rosls** is part of the `rosbash` suite. It allows you to **ls** directly in a package by name rather than by absolute path.

```
$ rosls [locationname[/subdir]]
```

For example:

```
$ rosls roscpp_tutorials
cmake launch package.xml srv
```

---

[1] `pwd` stands for Print Working Directory (shell builtin). The default action is to show the current folder as an absolute path. All components of the path will be actual folder names - none will be symbolic links.

Chapter 3

Creating a ROS Package

For a package to be considered a catkin package it must meet a few **requirements**:

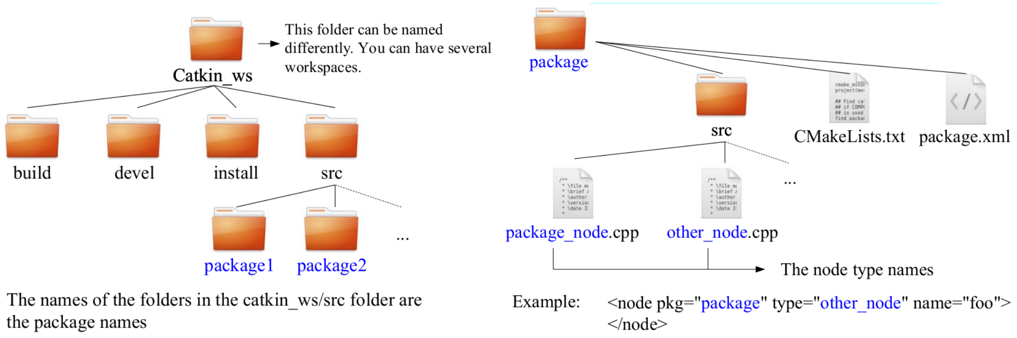
- The package must contain a catkin compliant **`package.xml`** file. That package.xml file provides meta information about the package.
- The package must contain a **`CMakeLists.txt`** which uses catkin. If it is a catkin metapackage it must have the relevant boilerplate CMakeLists.txt file.
- Each package must have its **own folder**. This means no nested packages nor multiple packages sharing the same directory.

The simplest possible package might have a structure which looks like this:

```
my_package/
    CMakeLists.txt
    package.xml
```

The recommended method of working with catkin packages is **using a catkin workspace**, but you can also build catkin packages standalone. A trivial workspace might look like this:

```
workspace_folder/ -- WORKSPACE
    src/ -- SOURCE SPACE
        CMakeLists.txt -- 'Toplevel' CMake file, provided by catkin
        package_1/
            CMakeLists.txt -- CMakeLists.txt file for package_1
            package.xml -- Package manifest for package_1
        ...
        package_n/
            CMakeLists.txt -- CMakeLists.txt file for package_n
            package.xml -- Package manifest for package_n
```



So, we have to use the **`catkin_create_pkg`** script to create a new catkin package. First, we have to change to the source space directory of the catkin workspace you created. Then, use the `catkin_create_pkg` script to create a new package called, for example, `'beginner_tutorials'` which **depends** on `std_msgs`, `roscpp`, and `rospy`:

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

This will create a `beginner_tutorials` folder which contains a `package.xml` and a `CMakeLists.txt`, which have been partially filled out with the information you gave `catkin_create_pkg`.

`catkin_create_pkg` requires that you give it a `package_name` and optionally a list of dependencies on which that package depends:

```
# This is a template example, do not try to run this:
# catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

`catkin_create_pkg` also has more advanced functionalities which are described in www./wiki.ros.org/catkin/commands/catkin_create_pkg.

After the creation, we need to **build the packages** in the catkin workspace:

```
$ cd ~/catkin_ws
$ catkin_make
```

After the workspace has been built it has created a similar structure in the devel subfolder as you usually find under `/opt/ros/$ROSDISTRO_NAME`.

As we already saw, to add the workspace to your ROS environment you need to source the generated setup file:

```
$ . ~/catkin_ws/devel/setup.bash
```

3.1 Package dependencies

First-order dependencies When using `catkin_create_pkg` earlier, a few package dependencies were provided. These first-order dependencies can now be reviewed with the **`rospack depends1`** tool.

```
$ rospack depends1 beginner_tutorials
roscpp
rospy
std_msgs
```

As you can see, `rospack` lists the same dependencies that were used as arguments when running the package creation script. These dependencies for a package are stored in the `package.xml` file. To visualize it in console use the `cat` command:

```
$ roscd beginner_tutorials
$ cat package.xml

<package format="2">
...
    <buildtool_depend>catkin</buildtool_depend>
    <build_depend>roscpp</build_depend>
    <build_depend>rospy</build_depend>
    <build_depend>std_msgs</build_depend>
...
</package>
```

Indirect dependencies In many cases, a dependency will also have its own dependencies. For instance, `rospy` has other dependencies:

```
$ rospack depends1 rospy
genpy
```

```
roscpp
rosgraph
rosgraph_msgs
roslib
std_msgs
```

A package can have quite a few indirect dependencies. Luckily **rospack depends** can **recursively** determine all **nested dependencies**:

```
$ rospack depends beginner_tutorials
cpp_common  rostime  roscpp_traits  roscpp_serialization  catkin  genmsg  genpy
message_runtime  gencpp  geneus  gennodejs  genlisp  message_generation  rosbuild
rosconsole  std_msgs  rosgraph_msgs  xmlrpcpp  roscpp  rosgraph  ros_environment
rospack  roslib  rospy
```

3.2 Package customization

At Section 6 of www.wiki.ros.org/ROS/Tutorials/CreatingPackage you can find all the instruction in order to custom your newly created package:

```
6.1 Customizing the package.xml
     6.1.1 description tag
     6.1.2 maintainer tags
     6.1.3 license tags
     6.1.4 dependencies tags
     6.1.5 Final package.xml
6.2 Customizing the CMakeLists.txt
```

3.3 Namespaces

Default node name To specify the name of the node in his source code we use:

```
int main (int argc, char** argv)
{
        //ROS Initialization
        ros::init(argc, argv, "some_name");
(...)
```

The default node name must not include any namespace (i.e. no “/” in the name).

The default node name is the only one used when the node is launched with:

```
$ rosrun package_name some_name
```

But when launched **from a launch file**, the **“name” field** is **compulsory**.

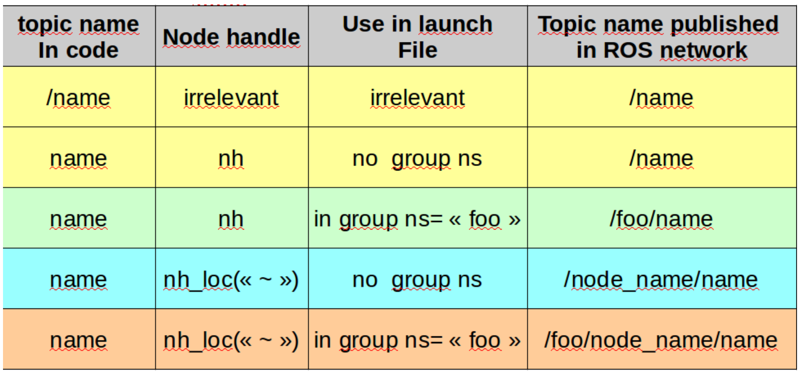
Topic name in code vs in network Disregarding remap instructions, which can always alter topic names, the name of a topic in the network depends on:

– Its name in the source code of the node, and whether it **starts with a “/”** or not.

– Whether the subscriber/publisher object is created using a **local namespace node handle**, e.g. nh(‘‘∼’’), or a **global namespace node handle**, e.g. nh.

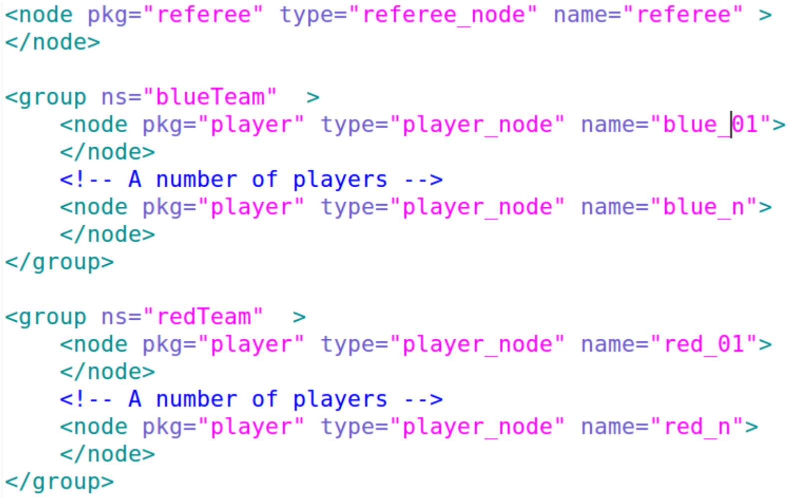
– Whether it is used within a <group ns=''...''> ... </group> in a launch file (**namespace**).

In fact, we can notice that from the following table:



topic name In code	Node handle	Use in launch File	Topic name published in ROS network
/name	irrelevant	irrelevant	/name
name	nh	no group ns	/name
name	nh	in group ns= « foo »	/foo/name
name	nh_loc(« ~ »)	no group ns	/node_name/name
name	nh_loc(« ~ »)	in group ns= « foo »	/foo/node_name/name

Example Imagine an application controlling two teams of players and a referee. The launch file could include:



```
<node pkg="referee" type="referee_node" name="referee" >
</node>

<group ns="blueTeam"  >
    <node pkg="player" type="player_node" name="blue_01">
    </node>
    <!-- A number of players -->
    <node pkg="player" type="player_node" name="blue_n">
    </node>
</group>

<group ns="redTeam"  >
    <node pkg="player" type="player_node" name="red_01">
    </node>
    <!-- A number of players -->
    <node pkg="player" type="player_node" name="red_n">
    </node>
</group>
```

Let us look at some topics that could be present in the application:

- The `whistle` topic could fall into the first/second case.
- The captain's instruction could be in the third case.
- The control topics for the players legs/feet it would be convenient to set in the fifth case.

Then:

- If `player_node` and `player_control_node` respectively subscribe to and publish to the topic `leg_control` in their code, with a local namespace handle and no initial "/", no remapping of this topic between these nodes is necessary.
- If `referee_node` and `player_node` publish to and subscribe to `/whistle`, no remapping is necessary for this topic.

Chapter 4

Run ROS nodes

Here we'll use the already seen lighweight simulator `ros-tutorials`:

```
$ sudo apt-get install ros-<distro>-ros-tutorials
```

Recall:
– Nodes: A node is an executable that uses ROS to communicate with other nodes.
– Messages: ROS data type used when subscribing or publishing to a topic.
– Topics: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.
– Master: Name service for ROS (i.e. helps nodes find each other)
– `rosout`: ROS equivalent of `stdout/stderr`
– `roscore`: Master + `rosout` + parameter server (parameter server will be introduced later)

A node really isn't much more than an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service.

ROS **client libraries** allow nodes written in different programming languages to communicate. For example:

- `rospy` = Python client library
- `roscpp` = C++ client library

4.1 The **`roscore`** command

The command **`roscore`** is the first thing you should run when using ROS:

```
$ roscore
```

Open up a new terminal, and let's use **`rosnode`** to see what running roscore did.[1] This command displays information about the ROS nodes that are currently running. The `rosnode list` command lists these active nodes:

```
$ rosnode list
/rosout
```

This showed us that there is only one node running: `rosout`. This is always running as it collects and logs nodes' debugging output.

The `rosnode info` command returns information about a specific node:

---

[1] When opening a new terminal your environment is reset and your `~/.bashrc` file is sourced. If you have trouble running commands like rosnode then you might need to add some environment setup files to your `~/.bashrc` or manually re-source them.

```
$ rosnode info /rosout
--------------------------------------------------------------------------
Node [/rosout]
Publications:
* /rosout_agg [rosgraph_msgs/Log]

Subscriptions:
* /rosout [unknown type]

Services:
* /rosout/get_loggers
* /rosout/set_logger_level

contacting node http://machine_name:54614/ ...
Pid: 5092
```

4.2 The `rosrun` command

The **`rosrun`** command allows you to use the package name to directly run a node within a package (without having to know the package path).

```
$ rosrun [package_name] [node_name]
```

For example, we can run the`turtlesim_node` contained in the `turtlesim` package to show the TurtleSim window:

```
$ rosrun turtlesim turtlesim_node
```

Then, typing in a new terminal:

```
$ rosnode list
/rosout
/turtlesim
```

One powerful feature of ROS is that you can **reassign names** from the command-line. After closing the "turtlesim" window (to stop the node) re-run it, but using a **Remapping Argument** to change the node's name:

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

Now, if we go back and use `rosnode` list[2]:

```
$ rosnode list
/my_turtle
/rosout
```

4.3 The `rosnode ping` command

We can the use **`rosnode ping`** to test that the node is up:

```
$ rosnode ping my_turtle
rosnode: node is [/my_turtle]
pinging /my_turtle with a timeout of 3.0s
xmlrpc reply from http://DLANZA-HP250G1:46339/ time=0.771046ms
```

---

[2] If you still see `/turtlesim` in the list, it might mean that you stopped the node in the terminal using `ctrl+C` instead of closing the window, or that you don't have the `$ROS_HOSTNAME` environment variable defined as described in Network Setup - Single Machine Configuration. You can try cleaning the `rosnode list` with: **`$ rosnode cleanup`**.

```
xmlrpc reply from http://DLANZA-HP250G1:46339/ time=1.283884ms
xmlrpc reply from http://DLANZA-HP250G1:46339/ time=1.255989ms
xmlrpc reply from http://DLANZA-HP250G1:46339/ time=1.008034ms
ping average: 1.079738ms
```

Chapter 5

ROS topics & messages

For this part we will need three terminals:

```
# on Terminal 1:
$ roscore

# on Terminal 2:
$ rosrun turtlesim turtlesim_node

# on Terminal 3:
$ rosrun turtlesim turtle_teleop_key
```

Now we can use the arrow keys of the keyboard to drive the turtle around.

The `turtlesim_node` and the `turtle_teleop_key` node are communicating with each other over a **ROS Topic**.

`turtle_teleop_key` is publishing the key strokes on a topic, while `turtlesim` subscribes to the same topic to receive the key strokes.

Let's use `rqt_graph` which shows the nodes and topics currently running:

5.1 ROS topics and the `rostopic` command

The `rostopic` tool allows you to get information about ROS topics. You can use the help option to get the available sub-commands for `rostopic`:

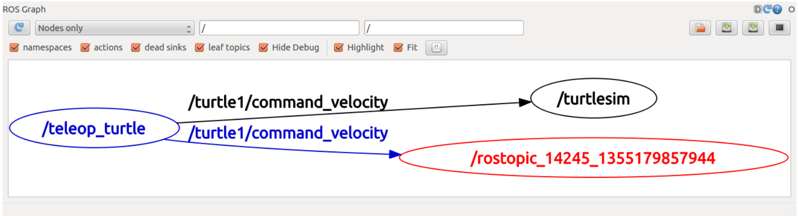
```
$ rostopic -h
rostopic bw display bandwidth used by topic
rostopic echo print messages to screen
rostopic hz display publishing rate of topic
rostopic list print information about active topics
rostopic pub publish data to topic
rostopic type print topic type
```

Let's use some of these topic sub-commands to examine turtlesim:

○ **`rostopic echo`** :

```
$ rostopic echo /turtle1/cmd_vel
# After pressing the left arrow button:
linear:
   x: 0.0
   y: 0.0
   z: 0.0
angular:
   x: 0.0
   y: 0.0
   z: -2.0
---
```

Note: if we look at `rqt_graph` again now, (after refresh) we can see `rostopic echo`, shown here in red, now also subscribed to the `turtle1/command_velocity` topic:



◦ **`rostopic list`** :

```
# Check the options with help command:
$ rostopic list -h
Usage: rostopic list [/namespace]
Options:
   -h, --help : show this help message and exit
   -b BAGFILE, --bag=BAGFILE : list topics in .bag file
   -v, --verbose : list full details about each topic
   -p : list only publishers
   -s : list only subscribers
   --host : group by host name

# Use the verbose one:
$ rostopic list -v
Published topics:
   * /turtle1/color_sensor [turtlesim/Color] 2 publishers
   * /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
   * /rosout [rosgraph_msgs/Log] 3 publishers
   * /rosout_agg [rosgraph_msgs/Log] 1 publisher
   * /turtle1/pose [turtlesim/Pose] 2 publishers
Subscribed topics:
   * /turtle1/cmd_vel [geometry_msgs/Twist] 2 subscribers
   * /rosout [rosgraph_msgs/Log] 1 subscriber
```

5.2 ROS messages and the `rostopic` command

Communication on topics happens by sending ROS messages between nodes. For the publisher (`turtle_teleop_key`) and subscriber (`turtlesim_node`) to communicate, the publisher and subscriber must send and receive the same type of message. This means that a topic type is defined by the message type published on it. The type of the message sent on a topic can be determined using:

∘ **rostopic type [topic]** (returns the message type of any topic being published):

```
$ rostopic type /turtle1/cmd_vel
geometry_msgs/Twist
```

We can also look at the details of the message using `rosmsg`:

```
$ 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
```

5.3 Other rostopic commands

∘ **rostopic pub [topic] [msg_type] [args]** publishes data on to a topic currently advertised:

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0,
                                                                          0.0, 1.8]'
```

∘ **rostopic hz [topic]** reports the rate at which data is published:

```
$ rostopic hz /turtle1/pose
subscribed to [/turtle1/pose]
average rate: 59.354
min: 0.005s max: 0.027s std dev: 0.00284s window: 58
average rate: 59.459
min: 0.005s max: 0.027s std dev: 0.00271s window: 118
```

5.4 Plotting

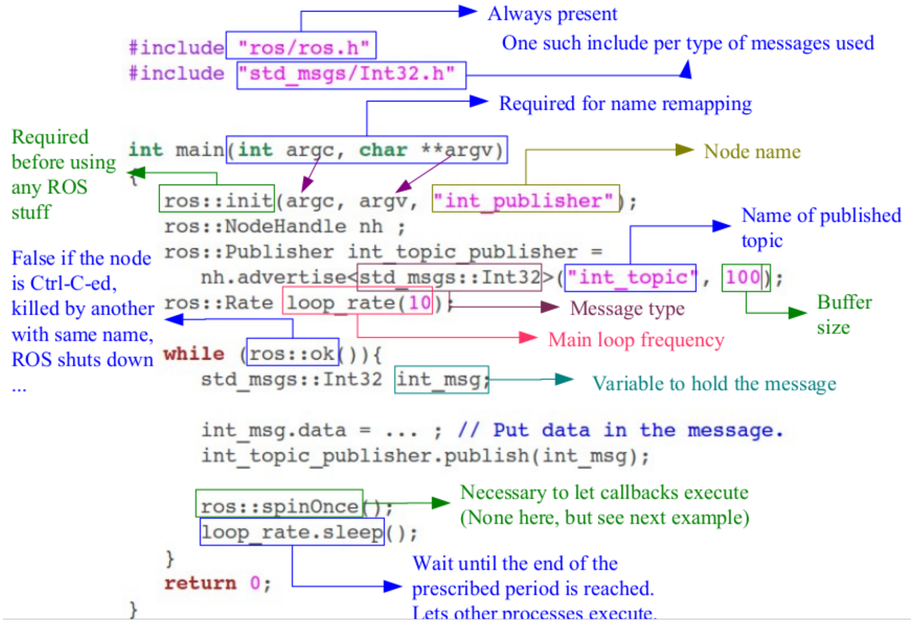
rqt_plot displays a scrolling time plot of the data published on topics. Here we'll use `rqt_plot` to plot the data being published on the `/turtle1/pose` topic.

Chapter 6

Subscribe & Publish

6.1 A simple C++ node

Here we have a first example in **C++** of a simple subscriber/publisher node that periodically publishes an integer:



```
#include "ros/ros.h"
#include "std_msgs/Int32.h"

int main(int argc, char **argv)
{
   ros::init(argc, argv, "int_publisher");
   ros::NodeHandle nh ;
   ros::Publisher int_topic_publisher =
      nh.advertise<std_msgs::Int32>("int_topic", 100);
   ros::Rate loop_rate(10);

   while (ros::ok()){
      std_msgs::Int32 int_msg;

      int_msg.data = ... ; // Put data in the message.
      int_topic_publisher.publish(int_msg);

      ros::spinOnce();
      loop_rate.sleep();
   }
   return 0;
}
```

This node uses no **subscriptions**, i.e. it has no inputs. Publishers can also subscribe to topics, which they use to produce their outputs (publish to topics).

A node which subscribes to a topic will be a little more complex, because it will **declare its subscription** in the initialization and it will have a **callback function** for each topic it subscribes to.

Callbacks will handle the data inputs “as though” asynchronously.

A “badly” written node Let’s see a publisher with a subscription in the following example:

```
/*
 *  This node subscribes to a turtle pose information and publishes a command velocity
 * which is same velocity as input, but inverted rotation speed.
 */

//Cpp
#include <sstream>
#include <stdio.h>
#include <vector>
#include <iostream>
#include <stdlib.h>

//ROS
#include "ros/ros.h"

//ROS msgs
#include <turtlesim/Velocity.h>
#include <turtlesim/Pose.h>

//Namespaces
using namespace std;

//Global variables
ros::Publisher pub_vel;

void poseCallback(turtlesim::Pose leader_pose){
    turtlesim::Velocity cmd_msg;

    // Reproduce speed and invert rotation speed
    cmd_msg.linear  =   leader_pose.linear_velocity;
    cmd_msg.angular = - leader_pose.angular_velocity;

    //Publish topic
    pub_vel.publish(cmd_msg);
}

int main (int argc, char** argv)
{
    //ROS Initialization
    ros::init(argc, argv, "turtle_control");
    ROS_INFO("Node turtle_control_node Connected to roscore");
    ros::NodeHandle nh_; //ROS Handler

    //Subscribing
    ROS_INFO("Subscribing to topics\n");
    ros::Subscriber pose_sub=nh_.subscribe
        <turtlesim::Pose> ("/turtle/pose", 1, poseCallback);

    //Publishing
    pub_vel = nh_.advertise
        <turtlesim::Velocity>("/turtle/command_velocity", 1);

    //Listen for topics
    ros::spin();

    ROS_INFO("ROS-Node Terminated\n");
}
```

Conveniently declares most
non topic-specific stuff

Each message used
requires a corresponding include.
Using `rostopic type`
informs you about the file to include.

The type of message
handled by this callback

Find out about these using:
rosmsg show turtlesim/Velocity

This object is a publisher of messages of type turtlesim::Velocity

Queue size

What to execute when a
message arrives

The (published) topic name.
Remapping always possible, so...

This node is written in a very simple form, in which **publishing is performed in the subscriber callback**. So, the data is published at the same rate it is received and there is no need for an input queue because the only input velocity we care about is the latest.

Even if this form is suitable for certain uses, the control has to execute at the rate of arrival of topics. Sometimes you may want it to be a lower rate.

Also, it becomes awkward when the control uses several input topics (in which callback do I execute the code?).

Moreover, there is a bad use of resources: the main loop runs at maximum frequency, possibly for nothing.

A better form :

```
//Namespaces
using namespace std;

//Global variables
ros::Publisher pub_vel;
turtlesim::Pose lastLeaderPose, lastFollowerPose ;

/*
 *     This node subscribes to two Pose informations, from the
 *    leader and follower turtles. The corresponding callbacks
 *    simply store these informations in global variables for
 *    the control algorithm to use them.
 */

void poseCallbackLeader(turtlesim::Pose leaderPose){
    lastLeaderPose = leaderPose ;
}

void poseCallbackFollower(turtlesim::Pose followerPose){
    lastFollowerPose = followerPose ;
}

int main (int argc, char** argv)
{
    //ROS Initialization
    ros::init(argc, argv, "turtleControl2");
    ROS_INFO("Node turtleControl Connected to roscore");
    ros::NodeHandle nh_;//ROS Handler

    //Subscribing
    ROS_INFO("Subscribing to topics\n");
    ros::Subscriber pose_sub_leader
        = nh_.subscribe<turtlesim::Pose>("leader_pose",1,poseCallbackLeader);
    ros::Subscriber pose_sub_follower
        = nh_.subscribe<turtlesim::Pose>("follower_pose",1,poseCallbackFollower);

    //Publishing
    pub_vel = nh_.advertise<turtlesim::Velocity>("follower_velocity", 1);

    ros::Rate rate(10);
    ROS_INFO("SPINNING @ 10Hz");
    while (ros::ok()){
        ros::spinOnce();
        // Control algorithm: move follower turtle towards leader turtle.
        ...

        cmd_msg.linear  = K1 * ... ;
        cmd_msg.angular = K2 * ... ;

        pub_vel.publish(cmd_msg);
        rate.sleep();
    }

    ROS_INFO("ROS-Node Terminated\n");
}
```

Here, the control will be at 10 Hz, independent of the frequency at which the turtle poses arrive.

Some useful tricks:

1. `rostopic list + grep` to find interesting topics, e.g.:
   `rostopic list | grep` command will display the topics with the word command. Very useful with complex systems like the Baxter (many topics)
2. I'm interested in topic `/turtle1/command_velocity`. What type are the corresponding messages?
   `rostopic type /turtle1/command_velocity`
3. ROS tells me type is: `turtlesim/Velocity`.
   I will need `#include <turtlesim/Velocity.h>`
4. What are the data fields of these messages?
   `rosmsg show turtlesim/Velocity`
5. ROS tells me:
   `float32 linear`
   `float32 angular`
6. Now I can fill a `turtlesim/Velocity` message...

A well configured IDE will alleviate your task a lot. For example, with `myVel` being an object of the `turtlesim::Velocity` class, when you type `myVel.`, the IDE will show that the '.' can be followed by either "linear" or "angular".

There is less need for `rosmsg show` commands. Instead of using `rosmsg show`, in the absence of a proper IDE, you can search `turtlesim/Velocity.h` over the internet. But it works only with ROS predefined message types.

Serious mistakes to avoid:

- **Defining subscribers/publishers before execution of `ros::init`**
  Reason: creating a subscriber/publisher requires communication with the `rosmaster`, which must be running first.
- **Defining subscribers/publishers within the main loop or some inner loop.**
  Creating a publisher/subscriber implies quite a bit of overhead. If you create a new one at each loop, it's never ready in time. Generates nasty, hard to debug problems.
- **Forgetting `ros::spinOnce` or executing it only in initialization.**
  Your callbacks do not execute. Usually your main loop relies on data gathered, and possibly processed by the callbacks.
- **Executing main loop even though no data has been collected by the callbacks yet.**
  The `ros::spinOnce()` function essentially means: check whether some data has arrived; if yes, process it. In the above example, the globals `LastLeaderPose` and `LastFollowerPose` may initially contain garbage. If necessary, use boolean variables to check that the callbacks have been executed.

Chapter 7

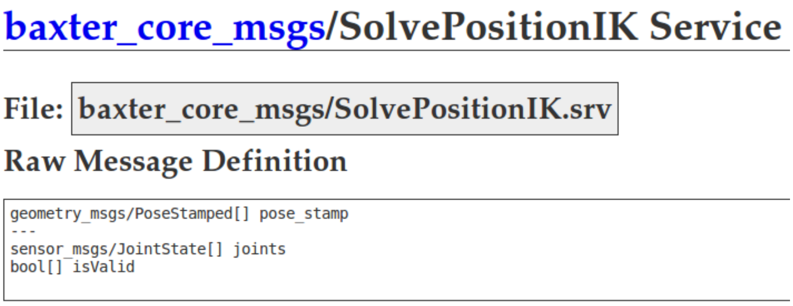
Services

A service is a way for a node to send a request and receive an answer in return. Services follow a client/server system, or request/response. They are analogous to **Remote Procedure Calls** (RPC).

The client sends a (strictly typed) request message and receives a (strictly typed) response message.

Example: calculate an inverse geometric model for a given pose of the end effector of a robot.

7.1 A sample service: Baxter IK



baxter_core_msgs/SolvePositionIK Service

File: baxter_core_msgs/SolvePositionIK.srv

Raw Message Definition

```
geometry_msgs/PoseStamped[] pose_stamp
---
sensor_msgs/JointState[] joints
bool[] isValid
```

The response uses an existing ROS message (`sensor_msgs/JointState`), of which only the position data makes sense.

7.2 Tips on using services

Initializations :

– Define the service client object before the main loop.

```
ros::ServiceClient some_client =
        nh_.serviceClient<somemessage::Type> ("serviceName");
```

– Check the existence of the service with "**exists**" method in the initialization part of the code. You have to wait until the service is ready since the node which provides the service may not yet be up and running. Alternatively, using "waitForService" is possible.

In the main loop :

– Always check the return value of the "call" method. It will be false if something went wrong.

```
if( !some_client.call(service_message) )
       ROS_ERROR(‘‘Call to service failed’’);
else
       // Do what you have to do...
```

– Carefully fill the request message. Improperly filled requests cause the call to fail.

– If the response message contains a validity field, check it after each call.

Using the service message object :

– Before the call, fill the request part

```
service_mesage.request = ?
```

– After the call, the result is in `service_message.response`.

– Alternatively, the call can take the form:

```
if( !some_client.call(request,response) )
       ROS_ERROR(?Call to service failed?);
else
{
       // Do what you have to do...
}
```

7.3 The `tf` package

The tf package helps you keep track of **transformations between frames**. The transformation between two frames can be calculated automatically as long as they belong to the same "**transformation tree**".

For a frame that you define to be in the transformation tree, you need to broadcast the transformation between your frame and a frame of the tree rooted at "world". It's done with a "**tf broadcaster**".

By default, the system assumes that frames move, so you need to periodically broadcast the position of a frame. After a certain amount of time during which a frame is not broadcast, the frame no longer appears in the transformation tree.

7.3.1 `tf` listener

The object which allows to obtain the transformation between two frames of a tree is a

```
tf::TransformListener listener;
```

Note: I never checked how much overhead there is when creating a `TransformListener` object, but I recommend defining it outside the main loop.

The listener is typically used within a try...catch structure.

```
try
{
       tf::StampedTransform desired_transform;
       listener.lookupTransform("frame1", "/frame2", ros::Time(0),
                                                  desired_transform);
       // Do something with the transform.
}
catch(tf::TransformException ex)
{
       ROS_ERROR("%s",ex.what());
```

```
        }
```

Chapter 8

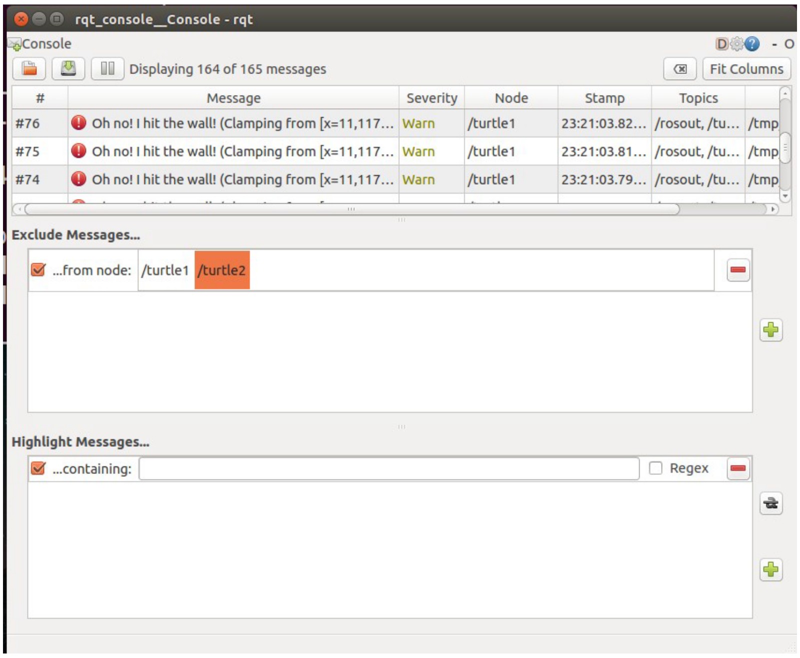
Various

8.1 ROS console output

Avoid the use of cout to print messages on the console. Use the ROS console! Messages at five different levels of verbosity:

```
Debug: ROS_DEBUG
Info: ROS_INFO
Warning: ROS_WARN
Error: ROS_ERROR
Fatal: ROS_FATAL
```

To visualize it use the `rqt_console`:



Displaying 164 of 165 messages

#	Message	Severity	Node	Stamp	Topics	
#76	Oh no! I hit the wall! (Clamping from [x=11,117...	Warn	/turtle1	23:21:03.82...	/rosout, /tu...	/tmp
#75	Oh no! I hit the wall! (Clamping from [x=11,117...	Warn	/turtle1	23:21:03.81...	/rosout, /tu...	/tmp
#74	Oh no! I hit the wall! (Clamping from [x=11,117...	Warn	/turtle1	23:21:03.79...	/rosout, /tu...	/tmp

☑ ...from node: /turtle1 /turtle2

☑ ...containing: ☐ Regex

8.2 ROS parameters

8.2.1 The parameter server

A shared dictionary accessible via network APIs. It is used to **store/retrieve parameters at runtime**. It does not have high performance, so better suited for **static data** (e.g. configuration parameters, tuning parameters, control gains,...)

Parameter types: 32 bit integers, Booleans, Strings, ...

There are global parameters and private parameters, specific to a node.

Example:

```
~/catkin_ws/src/turtle_control/launch/turtleControl.launch
```

8.2.2 Parameters from the command line

```
rosparam set <name> <value> : set parameter
rosparam get <name> : get parameter
rosparam load <file> : load parameters from file
rosparam dump <file> : dump parameters to file
rosparam delete <name> : delete parameter
rosparam list : list parameter names
```

Remarks:

- Avoid setting parameters from the command line. Do it in launch files preferably.
- Other commands are useful at check/debug time.
- Load/dump convenient to save/retrieve configurations.

8.2.3 Setting parameters in launch files

```
<node pkg="foopck" type="foopkg_node" name="foo1" cwd="node">
       <param name="foo_param" type="string" value="hello" />
</node>
```

If the `<param ...>` section is within a node section, then it concerns a parameter local to the node.

Several instances of the same node (with different names) can run with their own individual values of the parameters.

If the `<param ...>` section is not within a node section, it concerns a global parameter.

type="str—int—double—bool" (optional)

Specifies the type of the parameter. If you don't specify the type, `roslaunch` will attempt to automatically determine the type. These rules are very basic:

- Numbers with '.'s are floating point, integers otherwise;
- "true" and "false" are boolean (not case-sensitive).
- All other values are strings

8.2.4 Setting parameters in programs

`nh_` is the node handle (see previous sections). To access private parameters, the handle must be **created with the private namespace as its namespace**:

```
ros::NodeHandle nh_("~")
```

`ROS_INFO` outputs timed messages, `cout` does not. But it is not fundamental in this particular context.

8.3 Basics of `rosbag` (recording and replaying data)

Replaying data scenarios:

- Recording data for later replay in order to test/tune various algorithms is a very common need.
- The data need not be recorded by the same persons who will use the data.
- The same data may be shared with various teams working on similar problems.
- You need to be able to replay the data streams with “the same” timing as during the recording.
- You may use only some of the data (e.g. ignore some of the sensors or process only a particular time interval).