UNIVERSITÀ DEGLI STUDI DI TRENTO
Dipartimento di Ingegneria
e Scienza dell'Informazione

Fundamentals of Robotics
ROS programming – Part 2

Luigi Palopoli & Nicu Sebe & Michele Focchi

The catkin build system

- A build system is responsible for generating targets from source code
- Targets can be libraries, executable programs, generated scripts, exported interfaces.
- In ROS the source code is organised in packages, which typically consist of one or more targets after building
- Popular build systems
  - Gnu Make
  - Gnu Autotools
  - Cmake
  - Apache ant
- Most of the times build systems are integrated into IDE (e.g., eclipse, visual studio)

Why a build system for ROS?

- Standard build systems are effective but they are difficult to use when the complexity of the software grows
- Complexity generate dependencies and demand appropriate build rules
- ROS is made of loosely federated software components
  - Complex dependencies
  - Different language
  - Different code organisation conventions
- Development can become arduous
- *Catkin* embeds the dependencies and steers toward an effective conventional organisation of the code

Catkin installation

- In this class we will use catkin tools (a little different from the old versions)
- To install them, <u>for ubuntu users,</u> please type the following

```
$ sudo sh \
-c 'echo "deb http://packages.ros.org/ros/ubuntu `lsb_release -sc` main" \
> /etc/apt/sources.list.d/ros-latest.list'
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add –
$ sudo apt update
$ sudo apt install python3-catkin-tools
```

catkin

- Catkin generates executable, libraries and interfaces
- It offers an effective set of command line parameters
- Create a catkin workspace by:

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

- When you run it for the first time you will find the CMakeLists.txt in the src directory

catkin

- The catking work space contains three spaces
  - src : contains the source code. Create or clone your projects here
  - build: contains intermediate fie (do not touch)
  - devel: contains the completed build (awaiting installation)
  - install: if you called catkin_make install (useful for cross-compilation)
- After creating the workspace, and after any change, you have to use the source command to update the environment variables and functions

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

- If you now check

```
> echo $ROS_PACKAGE_PATH
```

Now you should have the catkin_ws directory in the path.

Catkin configuration

- catkin configuration can be checked through

```
> catkin config
```

```
luigi@ubuntu:~/catkin_ws$ catkin config
------------------------------------------------------------
Profile:                     default
Extending:          [cached] /opt/ros/noetic
Workspace:                   /home/luigi/catkin_ws
------------------------------------------------------------
Build Space:        [exists] /home/luigi/catkin_ws/build
Devel Space:        [exists] /home/luigi/catkin_ws/devel
Install Space:      [exists] /home/luigi/catkin_ws/install
Log Space:          [exists] /home/luigi/catkin_ws/logs
Source Space:       [exists] /home/luigi/catkin_ws/src
DESTDIR:            [unused] None
------------------------------------------------------------
Devel Space Layout:          linked
Install Space Layout:        merged
------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
Internal Make Job Server:    True
Cache Job Environments:      False
------------------------------------------------------------
Whitelisted Packages:        None
Blacklisted Packages:        None
------------------------------------------------------------
Workspace configuration appears valid.
------------------------------------------------------------
luigi@ubuntu:~/catkin_ws$
```

Creating catkin packages (from ROS tutorial)

- In order for a package to be catkin compliant it has to meet a few requirements
- The package must contain a **catkin compliant** package.xml file.
  - 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 has the structure:

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

Creating catkin packages

- We now show (following ROS tutorial) how to create a catkin compliant package.
- After creating a catkin workspace (see previous slides) let us move inside and create a package that depnds on std_msgs, rospy and rocpp.

```
> cd ~/catkin_ws/src
> catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

- The general form is

```
> cd ~/catkin_ws/src
> catkin_create_pkg package_name [dep1] [dep2] ..
```

Creating catkin packages

- After the package has been created I can be built in several ways
- The most “traditional” is

```
> cd ~/catkin_ws
> catkin_make
```

- Remember to add the work space after creation through the usual:

```
> source ~/catkin_ws/devel/setup.bash
```

- You can check the dependencies (both direct and indirect) through

```
> rospack depends1 beginner_tutorials
> rospack depends beginner_tutorials
```

Dependencies

```
> rospack depends1 beginner_tutorials
```

```
luigi@ubuntu:~/catkin_ws$ rospack depends1 beginner_tutorials
roscpp
rospy
std_msgs
luigi@ubuntu:~/catkin_ws$
```

```
> rospack depends beginner_tutorials
```

```
luigi@ubuntu:~/catkin_ws$ 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
luigi@ubuntu:~/catkin_ws$
```

Customising your package

- The call to catkin_create_pkg generates two files that allow you to make personalisations
- The first is package.xml and the second is CMakeLists.txt
- Both files are editable
- Let us start from package.xml

Customising your package

- The first lines of package.xml contain standard information about the file name, about the maintainer and the licensing policy
- They can be edited as suggested in the commented lines.

```xml
<?xml version="1.0"?>
<package format="2">
  <name>beginner_tutorials</name>
  <version>0.0.0</version>
  <description>The beginner_tutorials package</description>

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
  <maintainer email="luigi@todo.todo">luigi</maintainer>


  <!-- One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>
```

Dependencies

- Another segment of the package.xml that can be customised is related to the dependencies

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_export_depend>roscpp</build_export_depend>
<build_export_depend>rospy</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

- The personalisation of CMakeLists.txt is a longer story….

- Most of times it suffices to set:

```
<build_depend>package_name</build_depend>
<run_depend>package_name</run_depend>
```

ROS Launch

- The `roslaunch` facility allows us to launch multiple nodes (possibly on multiple hardwares) at once.
- The syntax is as simple as this:

```
>roslaunch [package] [filename.launch]
```

- Let us move to the package we just created and let us create a launch directory

```
> roscd beginner_tutorials
> mkdir launch
> cd launch
```

ROS Launch

- Now let us create a the following file and save it into turtletest.launch

```
<launch>
    <group ns="turtlesim1">
            <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
    </group>

    <group ns="turtlesim2">
            <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
    </group>

    <node pkg="turtlesim" name="mimic" type="mimic">
            <remap from="input" to="turtlesim1/turtle1/pose"/>
            <remap from="output" to="turtlesim2/turtle1/cmd_vel"/>
    </node>
</launch>
```

Create two groups with namespace tag turtlesim1 and turtlesim2.
Each has a turtlesim node named sim (no clash because of the namespaces).
Two simulators will start

We create and start a mimic node with its input and output topics renamed turtlesim1 and turtlesim2.
This way turtlesim2 will imitate turtlesim1

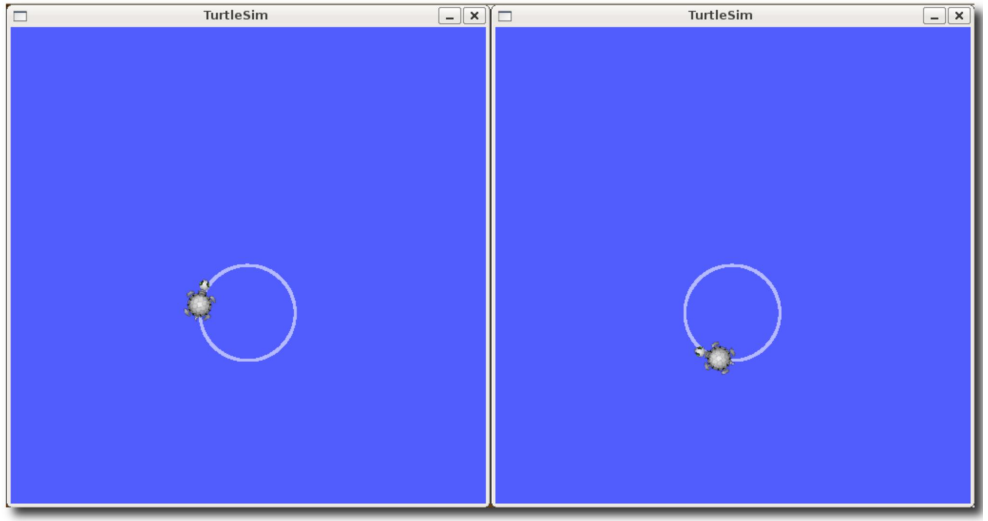
ROS Launch

- We can now launch by:

```
>roslaunch beginners_tutorial turtletest.launch
```

- And in a different terminal:

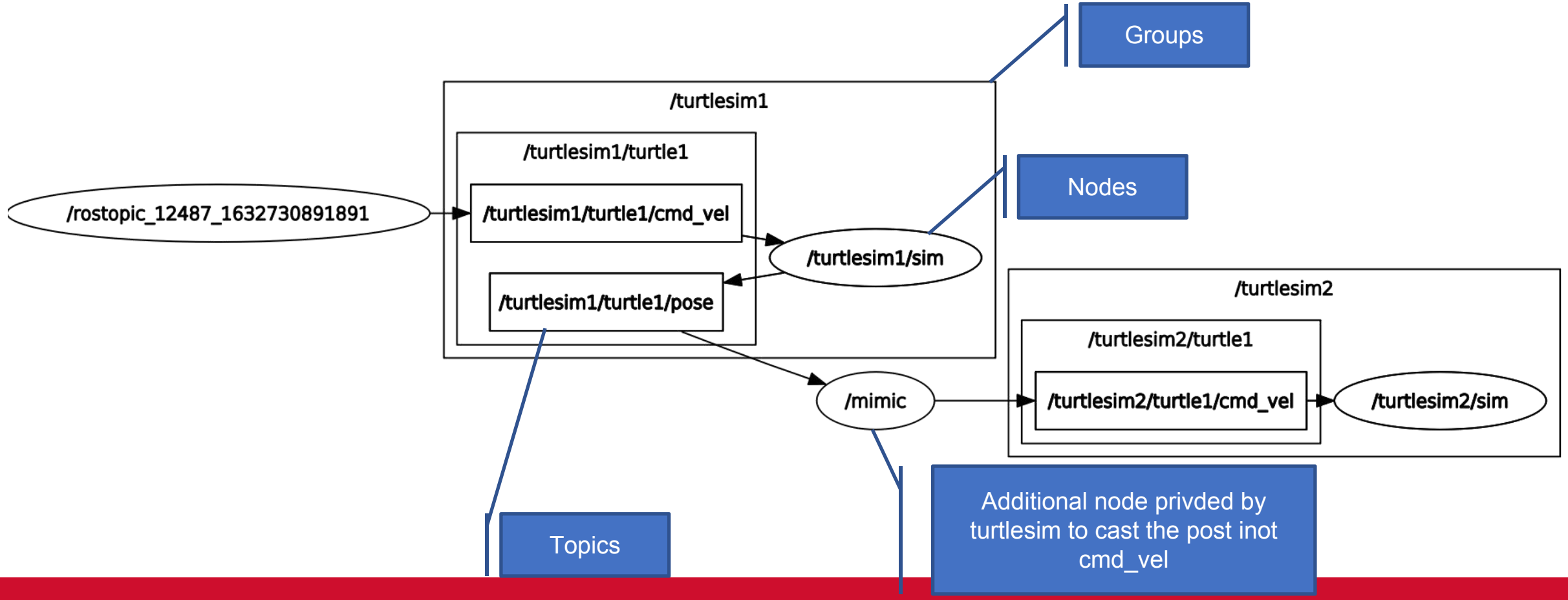
```
$ rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```



- We will have two simulations running in two different windows

What happend

- What happened is easily seen through rqt_graph



Changing the velocity

- As we have seen we can set the velocity for turtle1 and have it mimicked by turtle 2 by

```
$ rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[5.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

What happened?

- Let us check the type

```
>rostopic type /turtlesim1/turtle1/cmd_vel | rosmsg show
```

luigi@LAPTOP-S1KMH465: ~/catkin_ws/src/beginner_tutorials/launch

```
luigi@LAPTOP-S1KMH465:~/catkin_ws/src/beginner_tutorials/launch$ rostopic typ
e /turtlesim1/turtle1/cmd_vel | rosmsg show
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z

luigi@LAPTOP-S1KMH465:~/catkin_ws/src/beginner_tutorials/launch$
```

- We have a twist with 6 floats (3 for linear velocity and three for angular velocity

```
>rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1
-- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```

…and

- We publish a topic specifying the rate (-r) and the message content (--) using the YAML format

```
>rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1
-- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```

Editing Files

- rosed is part of the rosbash suite and allows us to edit directly a file within a package. Syntax

```
>rosed [package] [filename]
```

- Example

```
>rosed roscpp Logger.msg
```

- You can use smart tab for completion. The default editor is vim.
- you can change the default editor by defining and environment variable (which can be set in the .bashrc).

```
>export EDITOR="emacs"
```