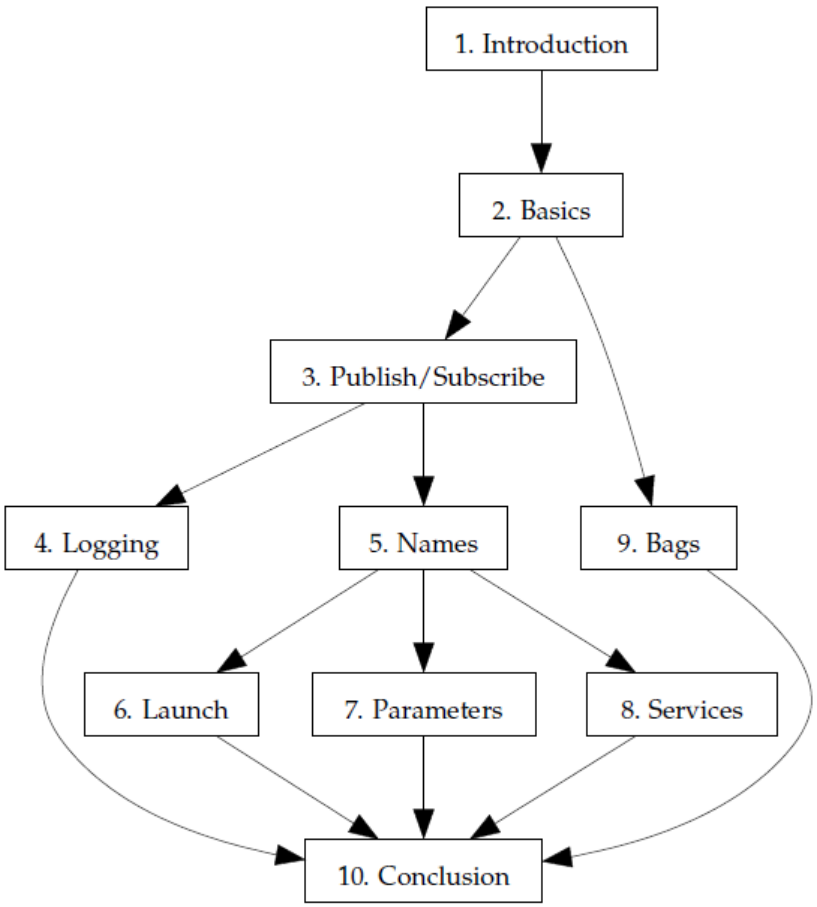
[Lab] Introduction to Robot Operating System (ROS)

Jae Yun JUN KIM*

September 28, 2020

Due: For next lecture/lab session (the 5 exercises asked in this document),

Reference: J. M. O'Kane, A Gentle Introduction to ROS, University of South Carolina, 2014.



1 A minimal example using turtlesim

Before we begin to examine the details of how ROS works, let's start with an example. This quick exercise will serve a few different purposes: it will help you confirm that ROS is installed

*ECE Paris Graduate School of Engineering, 37 quai de Grenelle CS71520 75 725 Paris 15, France; jae-yun.jun-kim@ece.fr

correctly, it will introduce the turtlesim simulator, and it will provide a working (albeit quite simple) system.
In three separate terminals, execute these three commands:

```
roscore
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
```

1.1 Packages

All ROS software is organized into packages. A ROS package is a coherent collection of files, generally including both executables and supporting files, that serves a specific purpose. In the example, we used two executables called turtlesim_node and turtle_teleop_key, both of which are members of the turtlesim package.

Listing and locating packages:

You can obtain a list of all of the installed ROS packages using this command:

```
rospack list
```

Each package is defined by a *manifest*, which is a file called package.xml. This file defines some details about the package, including its name, version, maintainer, and dependencies. The directory containing package.xml is called the *package directory*. (In fact, this is the definition of a ROS package: Any directory that ROS can find that contains a file named package.xml is a package directory.) This directory stores most of the package's files.
To find the directory of a single package, use the rospack find command:

`rospack find` *`package-name`*

For instance,

```
rospack find turtle
```

and, before pressing Enter, press the Tab key twice to see a list of all of the installed ROS packages whose names start with turtle.

Inspecting a package:

To view the files in a package directory, use a command like this:

`rosls` *`package-name`*

If you'd like to "go to" a package directory, you can change the current directory to a particular package, using a command like this:

`roscd` *`package-name`*

For instance,

```
rosls turtlesim
roscd turtlesim/images
```

1.2 The master

One of the basic goals of ROS is to enable roboticists to design software as a collection of small, mostly independent programs called nodes that all run at the same time. For this to work, those nodes must be able to communicate with one another. The part of ROS that facilitates this communication is called the ROSmaster. To start the master, use this command:

```
roscore
```

1.3 Nodes

Once you've started *roscore*, you can run programs that use ROS. A running instance of a ROS program is called a *node*. One node is an instance of an executable.

Starting nodes

The basic command to create a node (also known as "running a ROS program") is rosrun:

`rosrun` *`package-name executable-name`*.

There are two required parameters to rosrun. The first parameter is a package name. The second parameter is simply the name of an executable file within that package.

Listing nodes

ROS provides a few ways to get information about the nodes that are running at any particular time. To get a list of running nodes, try this command:

```
rosnode list
```

Note that the /rosout node is a special node that is started automatically by roscore. Its purpose is somewhat similar to the standard output (i.e. std::cout) that you might use in a console program.

Inspecting a node

You can get some information about a particular node using this command:

`rosnode info` *`node-name`*.

The output includes a list of topics for which that node is a publisher or subscriber, the services offered by that node, its Linux process identifier (PID), and a summary of the connections it has made to other nodes.
For instance,

```
rosnode info turtlesim
```

Killing a node

To kill a node you can use this command:

`rosnode kill` *`node-name`*.

For instance,

```
rosnode kill turtlesim
```

Unlike killing and restarting the master, killing and restarting a node usually does not have a major impact on other nodes; even for nodes that are exchanging messages, those connections would be dropped when the node is killed and reestablished when the node restarts.
To remove dead nodes from the list, you can use this command:

```
rosnode cleanup
```

1.4 Topics and messages

The primary mechanism that ROS nodes use to communicate is to send messages. Messages in ROS are organized into named topics. The idea is that a node that wants to share information will publish messages on the appropriate topic or topics; a node that wants to receive information will subscribe to the topic or topics that it's interested in. The ROSmaster takes care of ensuring that publishers and subscribers can find each other; the messages themselves are sent directly from publisher to subscriber.

1.4.1 Viewing the graph

This idea is probably easiest to see graphically, and the easiest way to visualize the publish subscribe relationships between ROS nodes is to use this command:

```
rqt_graph
```

In this name, the *r* is for ROS, and the *qt* refers to the Qt GUI toolkit used to implement the program. You should see a GUI, most of which is devoted to showing the nodes in the current system. In this graph, the ovals represent nodes, and the directed edges represent publisher-subscriber relationships.

1.4.2 Messages and message types

Listing a node

To get a list of active topics, use this command:

```
rostopic list
```

The topic list should, of course, be the same as the set of topics viewable in rqt_graph, but might be more convenient to see in text form.

Echoing messages

You can see the actual messages that are being published on a single topic using the rostopic command

`rostopic echo` *`topic-name`*

For instance,

```
rostopic echo /turtle1/cmd_vel
```

Measuring publication rates

There are also two commands for measuring the speed at which messages are published and the bandwidth consumed by those messages:

`rostopic hz` *`topic-name`*

`rostopic bw` *`topic-name`*

These commands subscribe to the given topic and output statistics in units of messages per second and bytes per second, respectively.

For instance,

```
rostopic hz /turtle1/cmd_vel
rostopic bw /turtle1/cmd_vel
```

Inspecting a topic

You can learn more about a topic using the *rostopic* info command:

`rostopic info` *`topic-name`*

For instance,

```
rostopic info /turtle1/color_sensor
```

Inspecting a message type

To see details about a message type, use a command like this

`rosmsg show` *`message-type-name`*

For instance,

```
rosmsg show turtlesim/Color
```

Publishing messages from the command line

Most of the time, the work of publishing messages is done by specialized programs. However, you may find it useful at times to publish messages by hand. To do this, use rostopic:

`rostopic pub -r rate-in-hz` *`topic-name message-type message-content`*

This command repeatedly publishes the given message on the given topic at the given rate. The final message content parameter should provide values for all of the fields in the message type, in order. Here's an example:

```
rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[2, 0, 0]' '[0, 0, 0]'

rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[0, 0, 0]' '[0, 0, 1]'
```

Understanding message type names

Like everything else in ROS, every message type belongs to a specific package. Message type names always contain a slash, and the part before the slash is the name of the containing package:

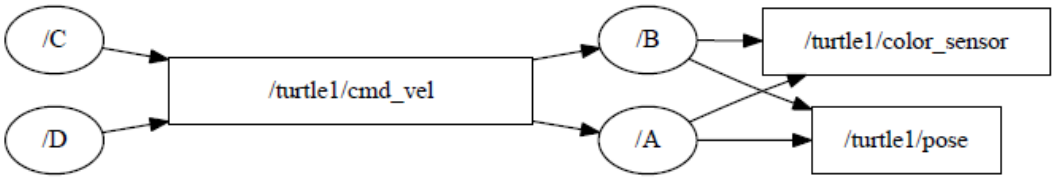
`package-name/type-name`

2 A larger example

So far in this chapter, we've seen how to start the ROS master, how to start ROS nodes, and how to investigate the topics those nodes use to communicate with one another. This section wraps up our introduction with an example a little larger.
First, stop any nodes that might be currently running. Then, do

```
roscore
rosrun turtlesim turtlesim_node __name:=A
rosrun turtlesim turtlesim_node __name:=B
rosrun turtlesim turtle_teleop_key __name:=C
rosrun turtlesim turtle_teleop_key __name:=D
```

This should start two instances of the turtlesim simulator. These should appear in two separate windows and two instances of the turtlesim teleoperation node. The only element in the example that might be unfamiliar is the __name parameter to rosrun. These parameters override the default name that each node tries to assign to itself. They're needed because the ROS master does not allow multiple nodes with the same name.



3 Writing ROS programs

So far we've introduced a few core ROS features, including packages, nodes, topics, and messages. We also spent a bit of time exploring some existing software built on those features. Now it's finally time to begin creating your own ROS programs.

3.1 Creating a workspace and a package

All ROS software, including software we create, is organized into packages. Before we write any programs, the first steps are to create a *workspace* to hold our packages, and then to create the package itself.

Creating a workspace

Packages that you create should live together in a directory called a *workspace*. Use the normal *mkdir* command to create a directory. We'll refer to this new directory as your workspace directory. One final step is needed to set up the workspace. Create a subdirectory called *src* inside the workspace directory. As you might guess, this subdirectory will contain the source code for your packages. Write the following commands in a terminal:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
```

Even though the workspace is empty (there are no packages in the 'src' folder, just a single CMakeLists.txt link) you can still "build" the workspace:

```
cd ~/catkin_ws
catkin_make
```

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.
Type the following command:

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

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

```
echo $ROS_PACKAGE_PATH
```

Creating a package

The command to create a new ROS package, which should be run from the *src* directory of your workspace, looks like this:

`catkin_create_pkg` *`package-name`*

It creates a directory to hold the package and creates two configuration files inside that directory.

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

- The first configuration file, called *package.xml*, is the manifest.
- The second file, called *CMakeLists.txt*, is a script for an industrial-strength cross-platform build system called CMake. It contains a list of build instructions including what executables should be created, what source files to use to build each of them, and where to find the include files and libraries needed for those executables. CMake is used internally by catkin.

This package is located within the workspace as follows:
Type the following commands in a terminal to create your first ROS package:

```
cd ~/catkin_ws/src
catkin_create_pkg hello std_msgs rospy
cd ~/catkin_ws
catkin_make
. ~/catkin_ws/devel/setup.bash
```

```
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
```

Observe that *package.xml* and *CMakeLists.txt* are generated with a similar content to the below.

In *package.xml*:

```
<?xml version="1.0"?>
<package>
  <name>hello</name>
<version >0.0.1</version>
<description>
Hello, ROS!
</description>
<maintainer email="">
</maintainer>
<license>TODO</license>
<buildtool_depend>catkin</buildtool_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>
</package>
```

In *CMakeLists.txt*:

```
# What version of CMake is needed ?
cmake_minimum_required (VERSION 2.8.3)

# Name of this package.
project(hello)

# Find the catkin build system , and any other packages on
# which we depend .
find_package(catkin REQUIRED COMPONENTS rospy std_msgs)

# Declare our catkin package.
catkin_package( )
```

```
# Specify locations of header files.
#include_directories(include ${catkin_INCLUDE_DIRS})

# Declare the executable , along with its source files. If
# there are multiple executables, use multiple copies of
# this line.
# add_executable (hello hello.cpp)

# Specify libraries against which to link. Again , this
# line should be copied for each distinct executable in
# the package .
#target_link_libraries(hello ${catkin_LIBRARIES})
```

3.2 Hello, ROS!

Now that we've created a package, we can start writing ROS programs in Python.
Type the following commands in a terminal to create your first ROS package:

```
cd ~/catkin_ws/src/hello
mkdir scripts
cd scripts
```

And, type the following python script using spyder:

```
#!/usr/bin/env python
# license removed for brevity
import rospy

if __name__ == '__main__':
    try:
        rospy.init_node('hello_ros', anonymous=True)
        rospy.loginfo("Hello, ROS!")
    except rospy.ROSInterruptException:
        pass
```

Save the file as *hello_script.py*.

Then, make it be executable:

```
chmod +x hello_script.py
```

I) Exercise:

By executing the python script, show in a terminal the message "Hello, ROS!"

3.3 Publisher and subscriber

Now by repeating the steps that you followed in the previous section, create another ROS project with the name "talker_listener" which should contain the following python scripts. This package should also depend on *rospy* and *std_msgs*.
In *talker.py*:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Then, make it be executable:

```
chmod +x talker.py
```

In *listener.py*:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
```

```
    listener()
```

Then, make it be executable:

```
chmod +x listener.py
```

II) Exercise:

Show that the `talker` correctly sends messages and that the `listener` correctly receives messages.

3.4 My turtle controller

III) Exercise:

A) Create a package named `my_turtle_control` that depends on rospy and geometry_msgs.

B) Write a python script that generates some random linear and angular speeds and publishes these speeds to the topic that is listened by a turtlesim node.

C) Show that the turtlesim node moves with the speeds specified by your controller.

Hint: Look at the C++ code given in Listing 3.4 from the ROS reference book (see the references section).

4 Launch files

If you've worked through all of the examples so far, by now you might be getting frustrated by the need to start so many different nodes, not to mention *roscore*, by hand in so many different terminals. Fortunately, ROS provides a mechanism for starting the master and many nodes all at once, using a file called a launch file. The use of launch files is widespread through many ROS packages. Any system that uses more than one or two nodes is likely to take advantage of launch files to specify and configure the nodes to be used. This chapter introduces these files and the *roslaunch* tool that uses them.

4.1 Using launch files

Let's start by seeing how roslaunch enables us to start many nodes at once. The basic idea is to list, in a specific XML format, a group of nodes that should be started at the same time. The following XML code shows a small example launch file that starts a turtlesim simulator, along with the teleoperation node that we saw previously and the subscriber node we wrote. This file is saved as example1.launch in the main package directory for the my_turtle_control package. Before we delve into specifics of the launch file format, let's see how those files can be used.
Write the following launch file and name it to be *example1.launch*.

```
<launch>
<node
pkg="turtlesim"
type="turtlesim_node"
name="turtlesim"
respawn="true"
/>
<node
pkg="turtlesim"
type="turtle_teleop_key"
name="teleop_key"
required="true"
launch-prefix="xterm -e"
/>
<node
pkg="my_turtle_control"
type="pubvel.py"
name="velocity_publisher"
output="screen"
/>
</launch>
```

Executing launch files

To execute a launch file, use the roslaunch command:

`roslaunch` *`package-name launch-file-name`*

You can invoke the example launch file using this command:

```
roslaunch my_turtle_control example1.launch
```

If everything works correctly, this command will start three nodes. You should get turtlesim window, along with another window that accepts arrow key presses for teleoperating the turtle. The original terminal in which you ran the roslaunch command should show the pose information logged by our subpose program. Before starting any nodes, roslaunch will determine whether roscore is already running and, if not, start it automatically.
An important fact about roslaunch — one that can be easy to forget — is that all of the nodes in a launch file are started at roughly the same time. As a result, you cannot be sure about the order in which the nodes will initialize themselves. Well-written ROS nodes don't care about the order in which they and their siblings start up.

Ending a launched session

To terminate an active roslaunch, use Ctrl-C. This signal will attempt to gracefully shut down each active node from the launch, and will forcefully kill any nodes that do not exit within a short time after that.

IV) Exercise:

By launching correctly the `example1.launch` file, show that you can control either using the keyboard or using your own python script.

4.2 Creating launch files

4.2.1 Where to place launch files

As with all other ROS files, each launch file should be associated with a particular package. The usual naming scheme is to give launch files names ending with .launch. The simplest place to store launch files is directly in the package directory. When looking for launch files, roslaunch will also search subdirectories of each package directory. Some packages, including many of the core ROS packages, utilize this feature by organizing launch files into a subdirectory of their own, usually called launch.

4.2.2 Basic ingredients

The simplest launch files consist of a root element containing several node elements.

Inserting the root element

Launch files are XML documents, and every XML document must have exactly one root element. For ROS launch files, the root element is defined by a pair of launch tags:

```
<launch>

   ...

</launch>
```

All of the other elements of each launch file should be enclosed between these tags.

Launching nodes

The heart of any launch file is a collection of node elements, each of which names a single node to launch. A node element looks like this:

<node
pkg="*package-name*"
type="*executable-name*"
name="*node-name*"
/>

A node element has three required attributes:

- The *pkg* and *type* attributes identify which program ROS should run to start this node. These are the same as the two command line arguments to rosrun, specifying the package name and the executable name, respectively.
- The *name* attribute assigns a name to the node. This overrides any name that the node would normally assign to itself in its call to rospy.init().

Directing output to the console

To override this behavior for a single node, use the out- put attribute in its node element:

```
output="screen"
```

Nodes launched with this attribute will display their standard output on screen instead of in the log files discussed above.

Requesting respawning

After starting all of the requested nodes, roslaunch monitors each node, keeping track of which ones remain active. For each node, we can ask roslaunch to restart it when it terminates, by using a respawn attribute:

```
respawn="true"
```

If you close the turtlesim window, the corresponding node will terminate. ROS quickly notices this and, since that node is marked as a respawn node, a new turtlesim node, with its accompanying window, appears to replace the previous one.

Requiring nodes

An alternative to respawn is to declare that a node is required:

```
required="true"
```

When a required node terminates, roslaunch responds by terminating all of the other active nodes and exiting itself. That sort of behavior might be useful, for example, for nodes that (a) are so important that, if they fail, the entire session should be abandoned, and (b) cannot be gracefully restarted by the respawn attribute. The example uses the required attribute for the turtle_teleop_key node. If you close the window in which the teleoperation node runs, roslaunch will kill the other two nodes and exit.

Launching nodes in their own windows

One potential drawback to using roslaunch, compared to our original technique of using rosrun in a separate terminal for each node, is that all of the nodes share the same terminal. This is manageable (and often helpful) for nodes that simply generate log messages, and do not accept console input. For nodes that do rely on console input, as turtle_teleop_key does, it may be preferable to retain the separate terminals. Fortunately, roslaunch provides a clean way to achieve this effect, using the *launch-prefix* attribute of a node element:

```
launch-prefix="command-prefix"
```

The idea is that roslaunch will insert the given prefix at the start of the command line it constructs internally to execute the given node. In example1.launch, we used this attribute for the teleoperation node:

```
launch-prefix="xterm -e"
```

Because of this attribute, this node element is roughly equivalent to this command:

```
xterm -e rosrun turtlesim turtle_teleop_key
```

As youmay know, the xterm command starts a simple terminal window. The -e argument tells xterm to execute the remainder of its command line (in this case, rosrun turtlesim turtle_teleop_key) inside itself, in lieu of a newinteractive shell. The result is that turtle_teleop_key, a strictly text-based program, appears inside a graphical window.

4.2.3 Launching nodes inside a namespace

The usual way to set the default namespace for a node— a process often called pushing down into a namespace — is to use a launch file, and assign the *ns* attribute in its node element:

```
ns="namespace"
```

Write the following launch file and name it to be *example2.launch*.

```
<launch>
<node
name="turtlesim_node"
pkg="turtlesim"
type="turtlesim_node"
ns="sim1"
/>
<node
pkg="turtlesim"
type="turtle_teleop_key"
name="teleop_key "
required="true"
launch-prefix="xterm -e"
ns="sim1"
/>
<node
name="turtlesim_node"
pkg="turtlesim"
type="turtlesim_node"
ns="sim2"
/>
<node
pkg="my_turtle_control"
type="pubvel.py"
name="velocity_publisher"
ns="sim2"
/>
</launch>
```

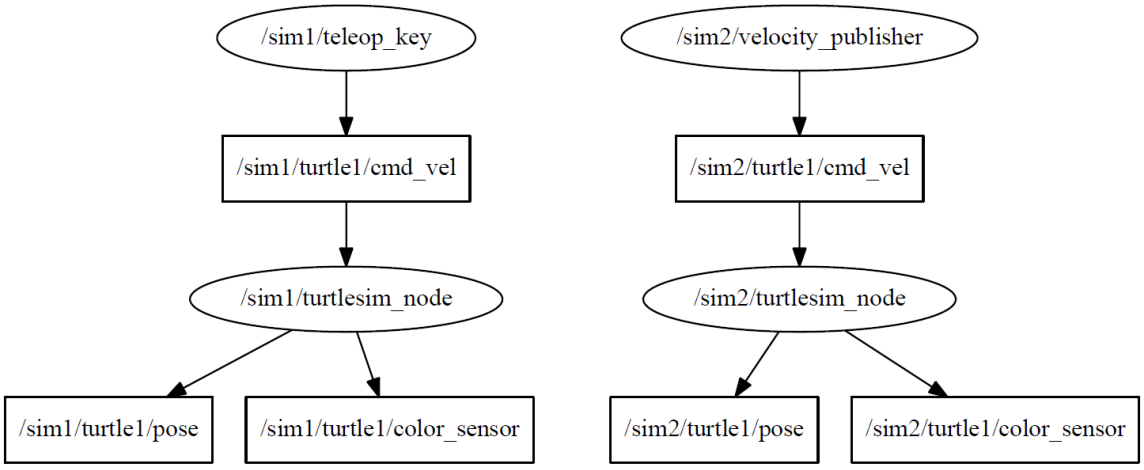
This example has some similarities to the system discussed in Section 4.1. In both cases, we start multiple turtlesim nodes. The results, however, are quite different. In Section 4.1, we changed only the node names, and left all of the nodes in the same namespace. As a result, both turtlesim nodes subscribe to and publish on the same topics. There is no straightforward way to interact with either of the two simulations individually. In the new example from Listing 4.2.3, we pushed each simulator node into its own namespace. The resulting changes to the topic names make the two simulators truly independent, enabling us to publish different velocity commands to each one.

V) Exercise:

By launching correctly the `example2.launch` file, show that you can control the first turtle using the keyboard and the second turtle using your own python script.

5 References

- The Python Tutorial, https://docs.python.org/3/tutorial/index.html
- J. M. O'Kane, A Gentle Introduction to ROS, University of South Carolina, 2014.


/sim1/teleop_key
/sim1/turtle1/cmd_vel
/sim1/turtlesim_node
/sim1/turtle1/pose
/sim1/turtle1/color_sensor
/sim2/velocity_publisher
/sim2/turtle1/cmd_vel
/sim2/turtlesim_node
/sim2/turtle1/pose
/sim2/turtle1/color_sensor