

Sonoma State University
School of Engineering Science

A Series of Lessons on Robotic Operating Systems

Olivia Piazza
Advisor: Dr. Sudhir Shrestha

Fall 2020

List of Contents

1	ROS: An Introduction	1
1.1	What is ROS?	1
1.2	The Outline of These Lesson Plans	1
2	Getting Started	2
2.1	Installing ROS	2
2.2	Configuring Your Environment	3
2.3	A Turtlesim Example!	4
2.4	Exploring Packages	4
2.5	On Master and Nodes	5
2.6	Topics and Messages	6
2.7	A More Complex Example	7
2.8	A Final Note	7
3	First Program	9
3.1	Your First Workspace and Package	9
3.2	A Publisher Program	11
3.3	A Subscriber Program	14
4	Log Messages	16
4.1	Log Message Levels	16
4.2	Viewing Log Messages	16
5	ROS Launch	18
5.1	Using Launch Files	18
5.2	The Anatomy of the Launch File	19
5.3	The End	19

1 ROS: An Introduction

1.1 What is ROS?

“The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

Why? Because creating truly robust, general-purpose robot software is hard. From the robot's perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments. Dealing with these variations is so hard that no single individual, laboratory, or institution can hope to do it on their own.

As a result, ROS was built from the ground up to encourage collaborative robotics software development. For example, one laboratory might have experts in mapping indoor environments, and could contribute a world-class system for producing maps. Another group might have experts at using maps to navigate, and yet another group might have discovered a computer vision approach that works well for recognizing small objects in clutter. ROS was designed specifically for groups like these to collaborate and build upon each other's work, as is described throughout this site. ” –ros.org

Robotics in the context of industry is about making a complicated tool to do a job. Like any manufacturing process, it has to meet certain performance requirements, expectations, and provide a good return on investment. As an engineer your job is to provide that and also ensure your robot can work in a safe manner. Major companies may or may not be interested in the framework you used to achieve these results, but knowing how robot systems are set up, how the programming and communication is done, and what the limitations of ROS are can be a nice box to check-off when applying for jobs. In the field of robotics research, knowing ROS is not only invaluable, it's expected.

Though the workings of ROS, specifically the C++ API, may take just a few weeks to master, the skills developed are transferable to other distributed systems with publication and subscription mechanisms. Once the basics are learned, you will have the skills to pursue more interesting topics such as motion planning, 3D perception, learning mechanisms and more.

1.2 The Outline of These Lesson Plans

This series of lessons is designed to give you a basic understanding of the main functions used in Robotic Operating Systems. It will begin by helping install the appropriate ROS onto your Ubuntu 20.04 host computer, providing a base vernacular for discussing elements of ROS. Next a section will be dedicated for writing your first ROS programs, including a package, publisher, and subscriber program. Log messages and graph resource names will briefly be discussed in the following lesson plans. The series will conclude with a final lesson on the ROS launch function. Each chapter is intended to be its own lesson plan and should take less than an hour to complete.

2 Getting Started

2.1 Installing ROS

ROS has many installation options to choose from including Kinetic Kame, Melodic Morenia, and Noetic Ninjemys (at the time of writing). Each version follows a naming convention which follows in alphabetic order of the previous version and is released approximately every two years (Noetic was released May 2020, whereas Melodic was released May 2018). Each version has long term support for some period of time, usually 5 years, and is recommended for specific operating systems. Because ROS Noetic is recommended for Ubuntu 20.04, I will be using this distribution. For previous versions of Ubuntu, such as 18.04, please refer to the ROS.org installation page.

To begin, you must configure your Ubuntu repository. As *root*, create a file called:

```
/etc/apt/sources.list.d/ros-latest.list
```

containing the single line:

```
deb http://packages.ros.org/ros/ubuntu focal main
```

Note, each version of Ubuntu is associated with a code name, in this case "focal", which can be found by typing the following in the command line:

```
lsb_release -a
```

Next, install the package authentication key. To do this, first download the key.

```
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key
```

If done correctly, you should have a small binary file called `ros.key`. Next you need to configure the apt package management system to use this key:

```
sudo apt-key add ros.key
```

Once this is done, the apt-key should say "OK", you can safely delete the `ros.key`.

Now that the repositories are configured, it is time to download the packages. First make sure everything is up to date:

```
sudo apt-get update
```

Next, we perform a complete install of the core ROS system:

```
sudo apt install ros-noetic-desktop-full
```

This command will install everything in **Desktop** plus 2D/3D simulators and 2D/3D perception packages. If you are pressed for disk-space, the ROS-base which is ROS packaging, build, and communication libraries, *no GUI tools*, can be installed instead:

```
sudo apt install ros-noetic-ros-base
```

A simple desktop install is also available which includes everything in the ROS-base plus basic visualization tools such as `rviz` and `rqt`:

```
sudo apt install ros-noetic-desktop
```

You can even install more packages available in ROS by using a command like

```
sudo apt install ros-noetic-<PACKAGE>
```

To see a list of available packages, the following command can be used:

```
apt search ros-noetic
```

At this point we are going to install one particular package which will be useful for us –turtlesim. Turtlesim is a simple simulation package and will be the basis for most examples in later lesson plans. If you installed Noetic using the desktop full command, turtlesim is already installed for you.

```
sudo apt-get install ros-noetic-turtlesim
```

After installing the ROS packages, we will need to perform a one-time initialization step, assuming we have done everything correctly thus far. First install rosdep:

```
sudo apt install python3-rosdep
```

Finally, initialize rosdep:

```
sudo rosdep init
```

2.2 Configuring Your Environment

Whether you have installed ROS on your personal computer or are using a computer on which ROS is already installed, there are two important configuration steps that must be done with every account. First you must initialize rosdep on your account:

```
rosdep update
```

This command stores some files in your home directory in a subdirectory called .ros. Next we need to setup a few environment variables to locate the files needed. To do this we run a bash script that ROS provides:

```
source /opt/ros/noetic/setup.bash
```

You should then confirm that everything was setup correctly:

```
export | grep ROS
```

If everything was setup correctly, you should see something like figure 1:

A terminal window screenshot showing the output of the command 'export | grep ROS'. The output lists several ROS environment variables that have been set, including ROSLISP_PACKAGE_DIRECTORIES, ROS_DISTRO, ROS_ETC_DIR, ROS_MASTER_URI, ROS_PACKAGE_PATH, ROS_PYTHON_VERSION, ROS_ROOT, and ROS_VERSION. The terminal prompt shows the user is in the directory ~/Noetic.

```
([redacted]@ubuntu:~/Noetic$ export | grep ROS
declare -x ROSLISP_PACKAGE_DIRECTORIES=""
declare -x ROS_DISTRO="noetic"
declare -x ROS_ETC_DIR="/opt/ros/noetic/etc/ros"
declare -x ROS_MASTER_URI="http://localhost:11311"
declare -x ROS_PACKAGE_PATH="/opt/ros/noetic/share"
declare -x ROS_PYTHON_VERSION="3"
declare -x ROS_ROOT="/opt/ros/noetic/share/ros"
declare -x ROS_VERSION="1"
[redacted]@ubuntu:~/Noetic$
```

Figure 1: Example of successful grep after installation and environment setup.

To configure this to run every terminal upon startup, we can add this bash script to our .bashrc file:

```
nano ~/.bashrc
source /opt/ros/noetic/setup.bash
```


Each package is defined by a **manifest** which is a file called package.xml. Each file contains details on the package: its name, version, maintainer, and dependencies. The directory containing package.xml is called the **package directory**, by definition.

Note, if your work space was built using catkin (which I did not show you) instead of rosbuilt, the previously outline structure is not true! Catkin uses a separate standardized directory hierarchy. For packages installed using apt-get, this hierarchy is rooted at `/opt/ros/noetic`, executables are stored in the lib subdirectory, include files are stored inside the include subdirectory, and ROS finds these files by search directories listed in CMAKE_PREFIX_PATH environment variable, setup by setup.bash.

The following is a list of useful commands related to packages. To find the directory of a single package, use the following:

```
rospack find <package_name>
```

To view files in a package directory:

```
rosls <package_name>
```

And finally, to go to the location of the package, use:

```
roscd <package_name>
```

Go ahead and try these commands out using turtlesim for the package name. What do you see? Pay close attention to the terminal's output, especially when using the rosls command.

One last thing on packages. On ROS forums, especially older ones, you might see the term **stacks** used. This term refers to a collection of related packages and is an outdated term. The new term for a stack is **Meta-Package** differs from a stack only in its file structure.

2.5 On Master and Nodes

So far we have only talked about files and how they are organized into packages. ROS software is developed as a collection of small, mostly independent programs called **nodes**. It is often necessary for nodes to communicate with each other. To facilitate this communication, ROS uses what's called a **ROS master**. We've already seen how to start the master before:

```
roscore
```

This command should be the first thing you type when you want to start a program or ensemble of programs and should not be stopped until you're finished using ROS. If it is stopped, connections between nodes will break and they cannot be reconnected –you will have to start over. If, however, you wanted new log files or to clear the parameter server, restarting roscore is the best way to do this.

As mentioned earlier, a running instance of a ROS program is called a **node**. To create a node, type rosrn:

```
rosls <package-name> <executable>
```

We did this in our example when we ran `rosls turtlesim turtlesim_node`. You could also call the node directly by typing out the full path: `/opt/ros/noetic/lib/turtlesim/turtlesim_node`, but this is a pain. To get a list of running nodes use the following:

```
rosls
```

To inspect a node:

```
roscallinfo <node-name>
```

To kill a node:

```
roscallinfo <node-name>
```

And to remove dead nodes:

```
roscallinfo cleanup
```

2.6 Topics and Messages

Messages are the primary mechanism ROS nodes use to communicate with each other. Messages are organized by **topics** and a node will **publish** a topic if it wants to share info. A node can also subscribe to a topic if it wants to receive info about the topic. To see a list of active topics the following can be used:

```
rostopic list
```

It is also possible to visualize topic structure using *rqt*. To do so, re-run your turtlesim example and in a fourth terminal type *rqt_graph*. For best results, change the settings to “Nodes/Topics(all)”, and disable all check-boxes except “Hide Debug”.

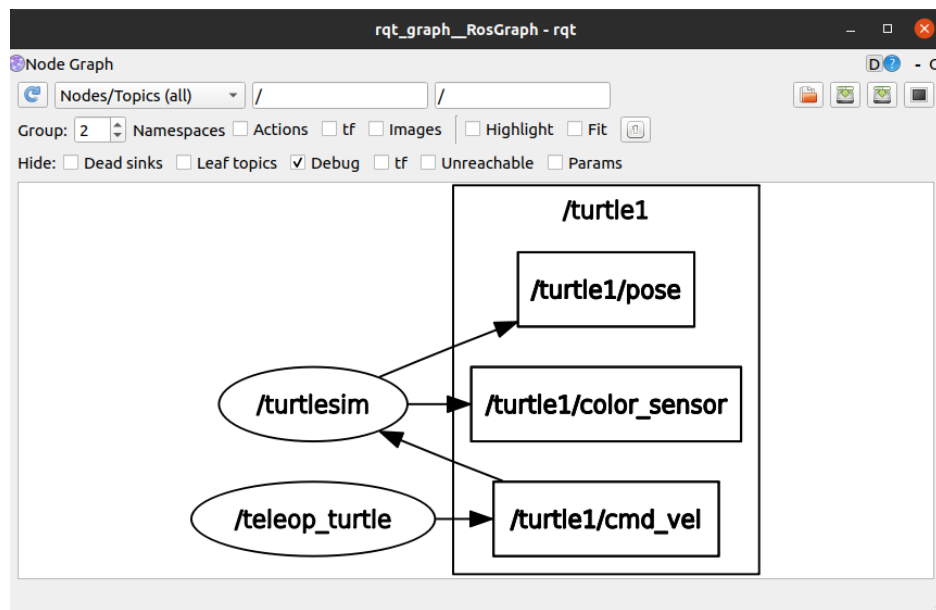


Figure 3: The *rqt* graph of our turtlesim example. Ovals represent nodes, directed edges represent publisher-subscriber relationships, and topics are shown as rectangles.

A few more commands are useful for messages and topics. We can echo messages using:

```
rostopic echo <topic-name>
```

We can also find the publication rate in messages per second:

```
rostopic hz <topic-name>
```

Or the bandwidth consumes:

```
rostopic bw <topic-name>
```


To inspect the topic:

```
rostopic info <topic-name>
```

Finally, to see details about the message type:

```
rosmmsg show <message-type-name>
```

We can also publish messages from the command line, though we usually automate this in our programs. To do this, we use `rostopic`:

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

Here's one for you to try. With your turtlesim example still open, type the following in a new terminal:

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

This message should drive the turtle straight ahead, along the x-axis with no rotation. Now try changing the parameter values around a bit. What do you get? Can you figure out what the two sets of parameters do? *Hint: each box [] corresponds to x,y,z values.*

You can also use a YAML file if you're familiar with that.

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist `linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
```

.

2.7 A More Complex Example

Now that we're getting comfortable with ROS commands, let's try a more complex example with the turtlesim package. Stop any nodes that are currently running. Next run `roscore` (if it's not already active). In four separate terminals type the following:

```
roslaunch turtlesim turtlesim_node __name:=A
roslaunch turtlesim turtlesim_node __name:=B
roslaunch turtlesim turtle_teleop_key __name:=C
roslaunch turtlesim turtle_teleop_key __name:=D
```

This should populate two instances of the turtlesim simulator in two separate windows. Next run the command `rqt_graph`. How many nodes are there? How many directed edges? Do you recognize all the topics? What are they?

2.8 A Final Note

A final word on troubleshooting. There is one more command line tool that is helpful when ROS is not behaving as expected and can be run with no arguments:

```
roswtf
```

This command will examine environment variables, installed files, running nodes, etc., to find whether there are any inconsistencies in `roscdep`, or if nodes have died, or if active nodes are correctly configured. `roswtf` will produce a report containing both warnings and errors based on the checks it performs. In general, warnings are something that seem odd but your program may still run just fine. Errors are problems you should address if you are experiencing problems.

For those of you who appreciate the naming convention, the online community will claim that it means “ROS Where’s The Fire”. If you do appreciate that one, you might also be interested in trying to figure out what *rosawesome* does. *Hint: its ROS functionality does less than the role it serves in the ROS community –to remind you that Morgan Quigley is awesome.*

3 First Program

3.1 Your First Workspace and Package

In the previous lesson we got familiar with how software is organized in ROS. Now we're going to write some of that software ourselves. Let's begin by making a workspace directory. To do this, use the command `mkdir` then whatever you want to name your workspace. For example my workspace was made using the following command:

```
mkdir ~/Noetic/workspace
```

Once this directory is made, we need to make a subdirectory named `src`:

```
cd ~/Noetic/workspace
mkdir src
```

Next, let's create a package. Go into your `src` directory using the command `cd src`, then use the following:

```
catkin_create_pkg hello
```

The command actually doesn't do much, but creates a directory to hold the package then creates two configuration files inside that directory, the first is our manifest, called *package.xml*, the second is a script containing build instructions called *CMakeLists.txt*.

Let's start by editing the manifest by going into our newly created "hello" directory and using the command `nano package.xml`:

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

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

  <!-- Url tags are optional, but multiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
```

If you've never seen XML files before, that's okay. We are going to interact with them very superficially. First note that each file has these symbols `<element>` some text `</element>`. These objects act as self-descriptors and are useful when we go to build our program. Also note that some elements look different: `<!-- element -->`. These are commented out elements. If you would like to use

them simply delete the "<!--" and "-->" from outside the brackets. There isn't much to do here so let's start by updating the maintainer email and name with our own information and comment out the license element. Your XML should look similar to this when you're done:

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

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

  <!-- Url tags are optional, but multiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
```

Now, let's write a simple C++ program and save it as hello.cpp in our src directory:

```
// This header is the standard ROS class
#include <ros/ros.h>

int main(int argc, char **argv) {

    //initialize
    ros::init(argc, argv, "hello_ros");

    // Establish this program as a ROS node
    ros::NodeHandle nh;

    //Send some output as a log message
    ROS_INFO_STREAM("Hello, ROS!");
}
```

We'll see how to compile this in a bit, but let's look at the code for just a moment. The header file, `ros/ros.h` is the standard ROS class and should be included in every ROS program you write (I can't think of a reason where this wouldn't be the case). The `ros::init` function initializes the ROS library and the last parameter is the default name of your node. The `ros::NodeHandle` object is the main mechanism through which our programs will interact with the ROS system. Finally, the `ROS_INFO_STREAM` generates an informational message and will be sent to the console screen. Now we need to update our `CMakeList.txt`. Find the `CMakeList` and edit it using the `nano` command.

Note, the auto-generated list is quite long and for the most part, you will only need to un-comment certain lines and perhaps edit them to include appropriate packages.

```
cmake_minimum_required(VERSION 3.0.2)
project(hello)
find_package(catkin REQUIRED COMPONENTS roscpp)
include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} hello.cpp)
target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES})
```

Again, all of these lines should be something you un-comment. You should not be typing much for this part of the exercise. Finally, let's go back to our manifest and update some dependencies by un-commenting the following line:

```
<depend>roscpp</depend>
```

Putting it all together, we build our workspace which links all the important files by going to our root workspace directory, /Noetic/workspace then typing:

```
catkin build
```

Note, if you run into trouble because python is already installed on the system, all you need to do is install the python dependencies. For example, I had to install/update two dependencies before catkin build worked: *pip install empy* from my activated conda environment and *sudo apt-get install --reinstall python-catkin-pkg*. Finally locate the catkin_pkg path using the command *locate catkin_pkg* and echo your current python path, *echo \$PYTHONPATH*. If these paths don't match, copy the path to locate catkin_pkg and in your .bashrc, add the line *export PYTHONPATH = \$PYTHONPATH:<your-result-from-locate-catkin_pkg>*. Save the file and run *source ~/.bashrc* to update your session. We can then source our files:

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

And finally we run everything in two separate terminals:

```
roscore
roslaunch hello hello
```

3.2 A Publisher Program

The next program demonstrates how to send commands to be published. The ros publisher has the following syntax:

```
ros::Publisher pub=node_handle.advertise <msg_type> (topic_name, queue_size);
```

Recall to find a topic name, use the command *rostopic list*. First let's move around our workspaces a bit by moving all the hello files into a new directory "Hello". Now we can start somewhat fresh. Create a src directory in the workspace, then from inside the src directory run the *"catkin_make_pkg <package-name>"* command. In this case, I used the package name "publisher". Don't forget this name, we'll need it when we build our files. Next let's create our C++ program file. Save the following code in a .cpp (within the src directory) with the name "publisher.cpp".

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <stdlib.h> // For rand() and RAND_MAX
```

```

int main(int argc, char **argv) {
    // Initialize ROS, create a node handle
    ros::init(argc, argv, "publisher");
    ros::NodeHandle nh;

    // Create a publisher object
    ros::Publisher pub = nh.advertise<geometry_msgs::Twist>
        ("turtle1/cmd_vel", 1000);

    // Seed the random number generator
    srand(time(0));

    // Loop at a 1Hz rate until the node is shut down.
    ros::Rate rate(1);
    while(ros::ok()) {
        // Create and fill in the message. the other four fields default to 0.
        geometry_msgs::Twist msg;
        msg.linear.x = double(rand())/double(RAND_MAX);
        msg.angular.z = 2*double(rand())/double(RAND_MAX) -1;

        // Publish the message.
        pub.publish(msg);

        // Send a message to rosout.
        ROS_INFO_STREAM("Sending random velocity command:"
            << " linear=" << msg.linear.x
            << " angular=" << msg.angular.z);

        // Wait until it's time for another iteration.
        rate.sleep();
    }
}

```

Now we should update our manifest to reflect who the maintainer is and also any build dependencies we might have. Specifically, we need to add the build dependencies for roscpp and geometry_msgs. After editing your manifest (package.xml), it should look similar to the following:

```

<?xml version="1.0"?>
<package format="2">
  <name>publisher</name>
  <version>0.0.0</version>
  <description>The publisher package</description>
  <maintainer email="olivia@sonoma.edu">olivia</maintainer>
  <depend>roscpp</depend>
  <depend>geometry_msgs</depend>
  <buildtool_depend>catkin</buildtool_depend>
</export>

```

```
</package>
```

Finally, let's tell CMakeLists.txt that it needs to look for certain packages. Update the find_package line and uncomment out the include, add_executable, and target_link_libraries like before:

```
cmake_minimum_required(VERSION 3.0.2)
project(publisher)
find_package(catkin REQUIRED COMPONENTS roscpp geometry_msgs)
catkin_package()
include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} publisher.cpp)
target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES})
```

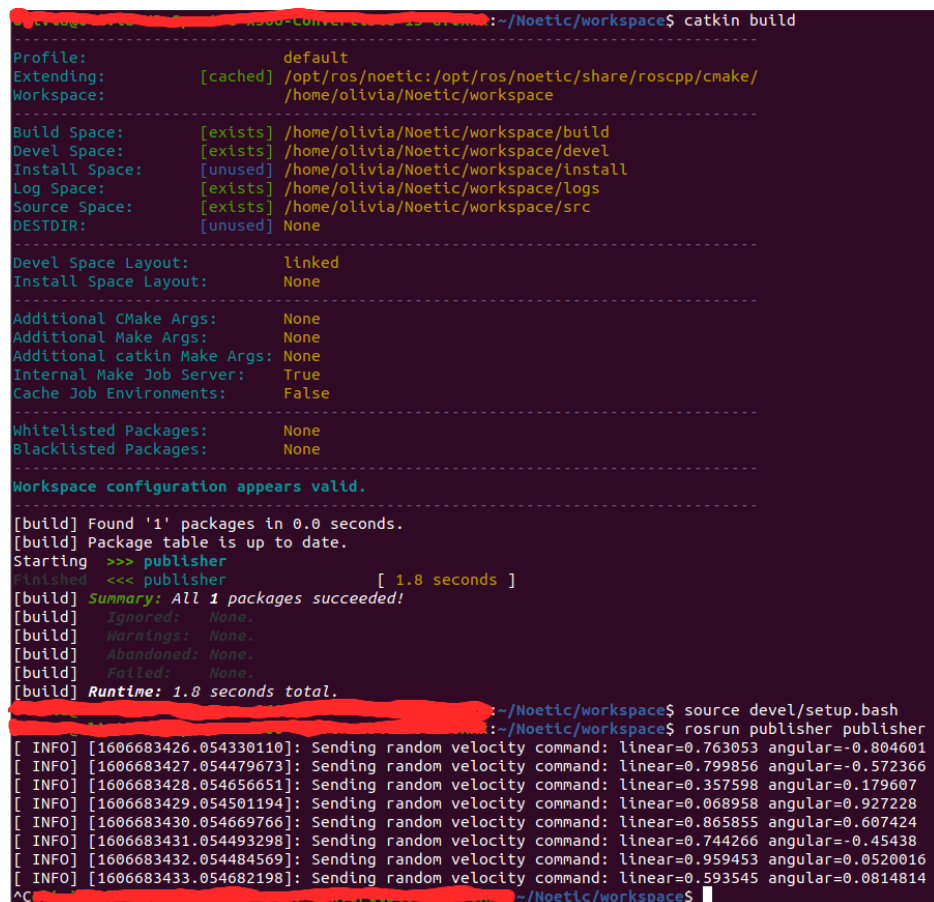
Alright, now that we have all the pieces together, let's build and source our project:

```
catkin build
source devel/setup.bash
```

After everything has been built, we can execute the publisher program using the following:

```
roscore
roslaunch publisher publisher
```

Figure 4 shows the output of catkin build as well as roslaunch.

A terminal window showing the output of catkin build and roslaunch. The catkin build output shows the workspace configuration, including build, devel, install, log, and source spaces, and the package table. The roslaunch output shows the publisher program sending random velocity commands to a roscore.

```
~/Noetic/workspace$ catkin build
Profile: default
Extending: [cached] /opt/ros/noetic:/opt/ros/noetic/share/ros/cpp/cmake/
Workspace: /home/olivia/Noetic/workspace

Build Space: [exists] /home/olivia/Noetic/workspace/build
Devel Space: [exists] /home/olivia/Noetic/workspace/devel
Install Space: [unused] /home/olivia/Noetic/workspace/install
Log Space: [exists] /home/olivia/Noetic/workspace/logs
Source Space: [exists] /home/olivia/Noetic/workspace/src
DESTDIR: [unused] None

Devel Space Layout: linked
Install Space Layout: None

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.

[build] Found '1' packages in 0.0 seconds.
[build] Package table is up to date.
Starting >>> publisher [ 1.8 seconds ]
[build] Summary: All 1 packages succeeded!
[build] Ignored: None.
[build] Warnings: None.
[build] Abandoned: None.
[build] Failed: None.
[build] Runtime: 1.8 seconds total.

~/Noetic/workspace$ source devel/setup.bash
~/Noetic/workspace$ roslaunch publisher publisher
[ INFO] [1606683426.054330110]: Sending random velocity command: linear=0.763053 angular=-0.804601
[ INFO] [1606683427.054479673]: Sending random velocity command: linear=0.799856 angular=-0.572366
[ INFO] [1606683428.054656651]: Sending random velocity command: linear=0.357598 angular=0.179607
[ INFO] [1606683429.054501194]: Sending random velocity command: linear=0.068958 angular=0.927228
[ INFO] [1606683430.054669766]: Sending random velocity command: linear=0.865855 angular=0.607424
[ INFO] [1606683431.054493298]: Sending random velocity command: linear=0.744266 angular=-0.45438
[ INFO] [1606683432.054484569]: Sending random velocity command: linear=0.959453 angular=0.0520016
[ INFO] [1606683433.054682198]: Sending random velocity command: linear=0.593545 angular=0.0814814
^C
```

Figure 4: Output of catkin build as well as the publisher program.

3.3 A Subscriber Program

Finally the last program we're going to write is a subscriber program. A subscriber doesn't know when a message will arrive so we'll have to use a callback function of the form:

```
void function_name(const package_name::type_name &msg) {  
    ...  
}
```

We also need to create a subscriber object to access our callback function:

```
ros::subscriber sub=node_handle.subscribe(topic_name, queue_size,  
                                           &pointer_to_callback_function)
```

Move all our old code into a directory called "publisher", then create and go into a new src directory from the workspace directory. Run the `catkin_create_pkg` command and call the project "subscriber". Now let's make our program and save it as `subscriber.cpp` in the `src` folder.

```
#include <ros/ros.h>  
#include <turtlesim/Pose.h>  
#include <iomanip>  
  
// The callback function  
void poseMessageReceived(const turtlesim::Pose& msg) {  
    ROS_INFO_STREAM(std::setprecision(2) << std::fixed  
                    << "position=(" << msg.x << ", " << msg.y << ")"  
                    << " direction=" << msg.theta);  
}  
  
int main(int argc, char **argv) {  
    // Initialize the ROS system and make a node  
    ros::init(argc, argv, "subscriber");  
    ros::NodeHandle nh;  
  
    // Create subscriber object  
    ros::Subscriber sub=nh.subscribe("turtle1/pose", 1000,  
                                     &poseMessageReceived);  
  
    // Let ROS take over  
    ros::spin();  
}
```

Again it is necessary to edit the manifest and CMakeList.

```
cmake_minimum_required(VERSION 3.0.2)  
project(subscriber)  
find_package(catkin REQUIRED COMPONENTS roscpp geometry_msgs)  
catkin_package()  
include_directories(include ${catkin_INCLUDE_DIRS})  
add_executable(${PROJECT_NAME} subscriber.cpp)  
target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES})
```



```
<depend>roscpp</depend>
<depend>geometry_msgs</depend>
<depend>turtlesim</depend>
```

Do the usual things to make the program run. From the workspace, run the following (with the `roscore` and `roslaunch` in three separate terminals):

```
catkin build
source devel/setup.bash
roscore
roslaunch turtlesim turtlesim_node
roslaunch subscriber subscriber
```

And there you have it! You've just made three programs that interact with ROS in very useful ways. As you learn more about ROS and its functionalities, these programs will serve as good references for some of the detail structure and build steps that are often glossed over in online tutorials or forums.

4 Log Messages

4.1 Log Message Levels

There are many levels of severity in ROS logs. The **DEBUG** level of message may be generated very frequently, but are almost never interesting when the program is working as it should. The **FATAL** message, however, tend to be of the utmost importance. Though rare, they indicate that there exists a problem that is preventing the program from continuing. The **INFO**, **WARN**, **ERROR** labels all represent severity levels in between DEBUG and FATAL. We can generate these messages using the following syntax:

```
ROS_<Severity-Level>_STREAM(message);
```

If we only want the message to print once, we can append the syntax with the `_ONCE` tag:

```
ROS_<Severity-Level>_STREAM_ONCE(message);
```

And if we wanted to control the rate at which we received these messages, the `_THROTTLE` syntax proves useful:

```
ROS_<Severity-Level>_STREAM_THROTTLE(interval, message);
```

We can even write a quick program that shows us how all these log messages work!

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

int main(int argc, char **argv) {
    // Initialize the ROS system
    ros::init(argc, argv, "log_severity_types");
    ros::NodeHandle nh;

    // Generate log messages of different severity levels
    ROS_DEBUG_STREAM("This is the Debug Message");
    ROS_INFO_STREAM("...and the Info Message");
    ROS_WARN_STREAM("...and now the Warn Message");
    ROS_ERROR_STREAM("...the Error Message");
    ROS_FATAL_STREAM("FATAL!");

    while(ros::ok()) {
        ROS_INFO_STREAM_ONCE("This message will only ever appear once.");

        ROS_INFO_STREAM_THROTTLE(2,
            "This message should appear every 2 seconds.");
    }
}
```

4.2 Viewing Log Messages

Log messages can be useful for debugging and communicating information to the user. We know how to create them, but where do these messages actually go? The answer to this is, a few places. First and probably easiest to access, the log message can appear as output on the console. To do this we can redirect the DEBUG and INFO streams to a file like this:

```
command > file.txt
```

For WARN, ERROR, and FATAL, their messages are sent to the standard error instead of the standard output, so we need to modify the command a bit:

```
command &> file.txt
```

Sometimes the log messages get out of order, so to ensure the two streams are in order, we can force the messages into the same buffering:

```
stdbuf -oL command &> file.txt
```

Finally to see the contents of the file, use the command `less -r file.txt`.

The next way to see log messages is to examine the topic `/rosout`. Since `/rosout` is just an ordinary topic that these messages get published to, you could the following to extract the message:

```
rostopic echo /rosout
```

Even more simple, you could invoke the GUI by using the following command:

```
rqt_console
```

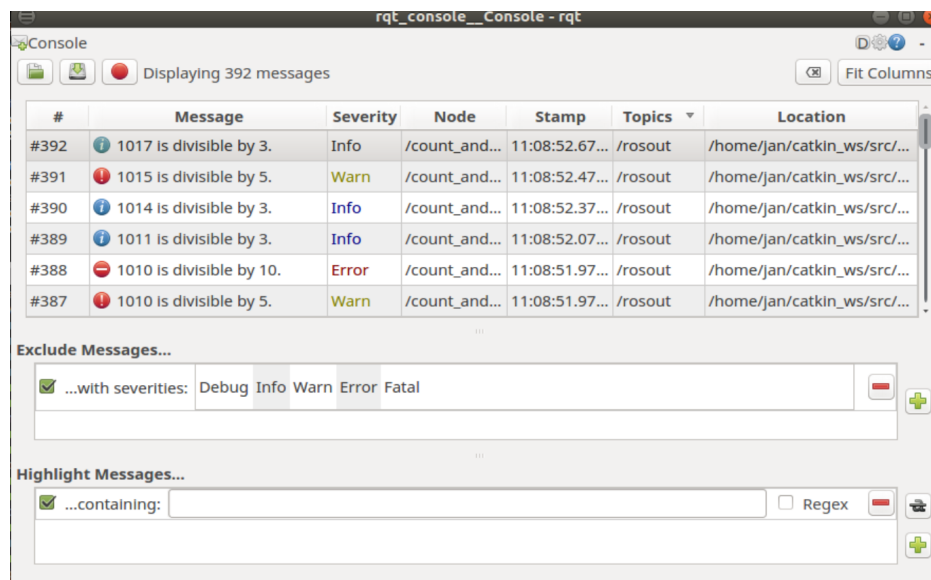


Figure 5: Example of the log messages using `rqt_console`.

Finally, log files get written to:

```
~/ .ros/log/run_id/rosout.log
```

Each `run_id` is unique and is based upon the MAC address and current time. To find the `run_id`, use the command `rosparam get /run_id`. Finally, log files accumulate over time, so to check how much disk space is being consumed by these logs, you can use the command `rosclean check` and when you're ready to delete them, `rosclean purge`. You can also delete the log files by hand, if you prefer.

5 ROS Launch

By now, you've worked through enough examples to know that starting and stopping many different nodes, not to mention roscore, every time is pretty annoying. Lucky for us there's a ROS mechanism for starting the master and many nodes all at once! This mechanism is called a **launch file** and is widespread among the ROS community. They're pretty useful, so let's take a look at how to use them and how they work.

5.1 Using Launch Files

The basic idea of launch files is that we can create an XML format file that a group of nodes can be started from at the same time. Remember that turtlesim simulator we started and stopped and restarted in the "Getting Started" lesson? Let's write a launch file to take care of starting all the nodes for us:

```
<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="subscriber"
    type="subscriber"
    name="subscriber"
    output="screen"
  />
</launch>
```

We save this file as `example.launch` and now we can invoke this launch file using the following command:

```
roslaunch subscriber example.launch
```

So where do we place these files? As with all other ROS files, each launch file should be associated with a particular package and the simplest place to store them is directly in the package directory. When looking for launch files, roslaunch will search subdirectories of each package directory. Some packages including core ROS packages even make their own launch subdirectory and store them in there.

At the heart of it, launch files are only made up of a few lines of code:

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

The example above is the bare minimum required to launch a node. The `pkg` and `type` attributes identify which program ROS should run to start the node. This is the same as running `roslaunch package-name executable-name`. The `name` attribute assigns a name to the node and overrides any name the node would normally assign to itself when we call `ros::init`.

5.2 The Anatomy of the Launch File

As already mentioned, a node element has three required attributes: `pkg`, `type`, and `name`. There were a few other attributes that were used in the example that could use some explaining. On the first node, the `turtlesim_node`, we used an attribute:

```
respawn="true"
```

After starting all the requested nodes, `roslaunch` monitors each node, tracking which ones are running and which have been terminated. In this case, we asked `roslaunch` to restart this particular node if it is terminated using the `respawn` attribute. An alternative to `respawn` is to declare that a node is required:

```
required="true"
```

When a required node is terminated, `roslaunch` terminates all other nodes, then exits. This behavior could be preferable, but it very case dependent. For our launch file, it makes sense to make the screen that displays our turtle pip respawn, but if the teleop key controller gets terminated all programs get terminated.

Another command used was the *launch-prefix* command. This command is used to launch nodes in their own window and the syntax is as follows:

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

As you may or may not know, `xterm -e` is a command that starts a simple terminal window, with the `-e` argument telling `xterm` to execute the remainder of its command line, in this case, `roslaunch turtlesim turtle_teleop_key`. The `launch-prefix` isn't limited to `xterm`, you can also use it for debugging, with the argument *gdb*, or for lowering scheduling priority of a process via *nice*.

5.3 The End

And that's about it. You now have been exposed to the basics of Robotic Operating Systems. There is much you can do now, and you're not limited to using C++ for developing in this language either (see `rospy` packages for more information). From here you should be able to understand ROS tutorials hosted on the `ros.org` website, the go-to authority on all things ROS and the suggested next stop if you wish to continue your studies in the field of robotics.