ROS.org

https://vimeo.com/639236696

ROS Introduction 3:12 (captioned) **PLAY THIS**

Open Robotics
Mountain View, CA, USA
We help make Robot Operating System (ROS) and the Ignition/Gazebo simulator.
info@openrobotics.org
Homepage
http://www.openrobotics.org
http://www.ros.org

An operating system is a software that provides interface between the applications and the hardware.

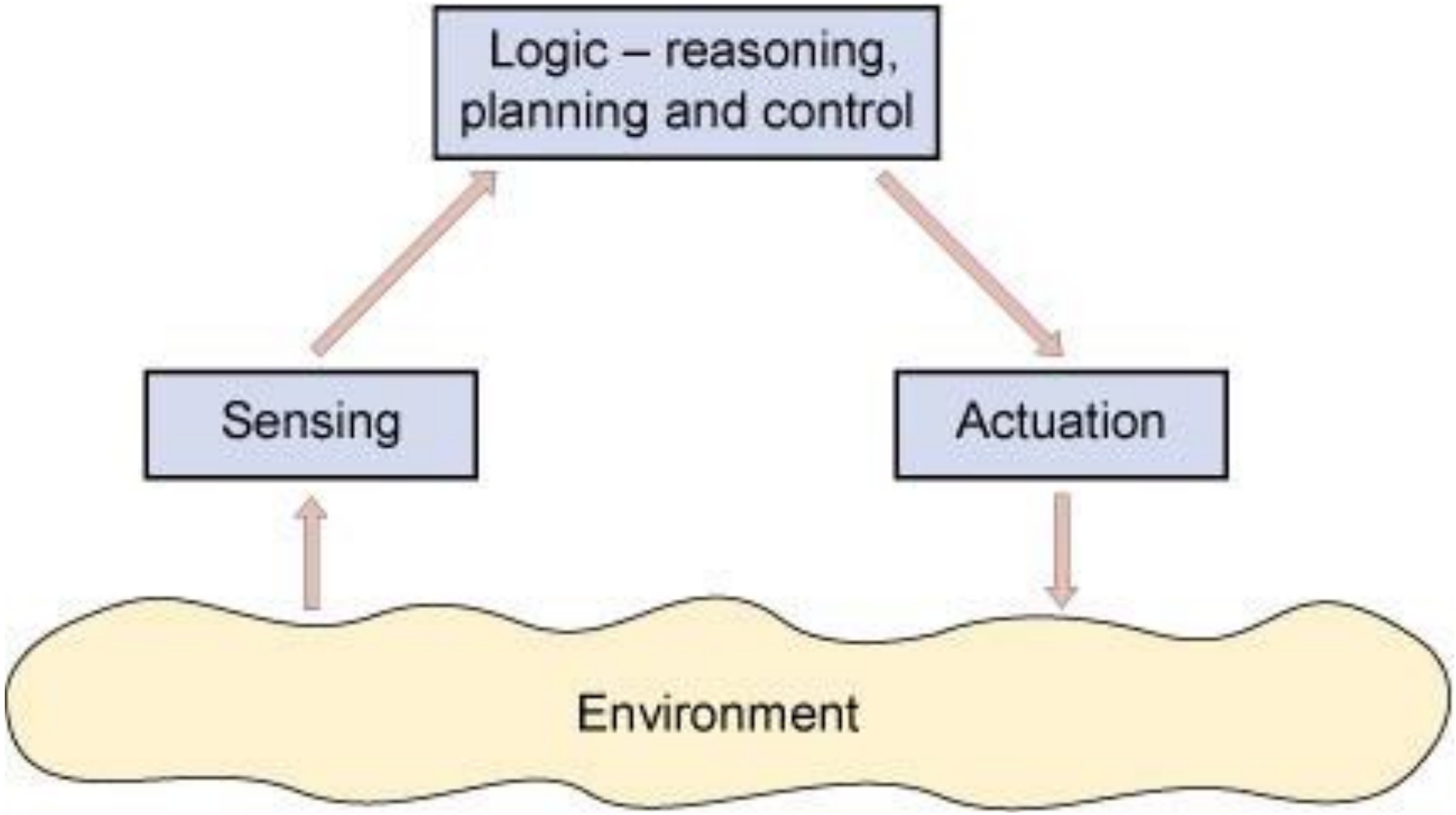
It deals with the allocation of resources such as memory, processor time etc. by using scheduling algorithms and keeps record of the authority of different users, thus providing a security layer.

The operating systems may include basic applications such as web browsers, editors, system monitoring applications etc.

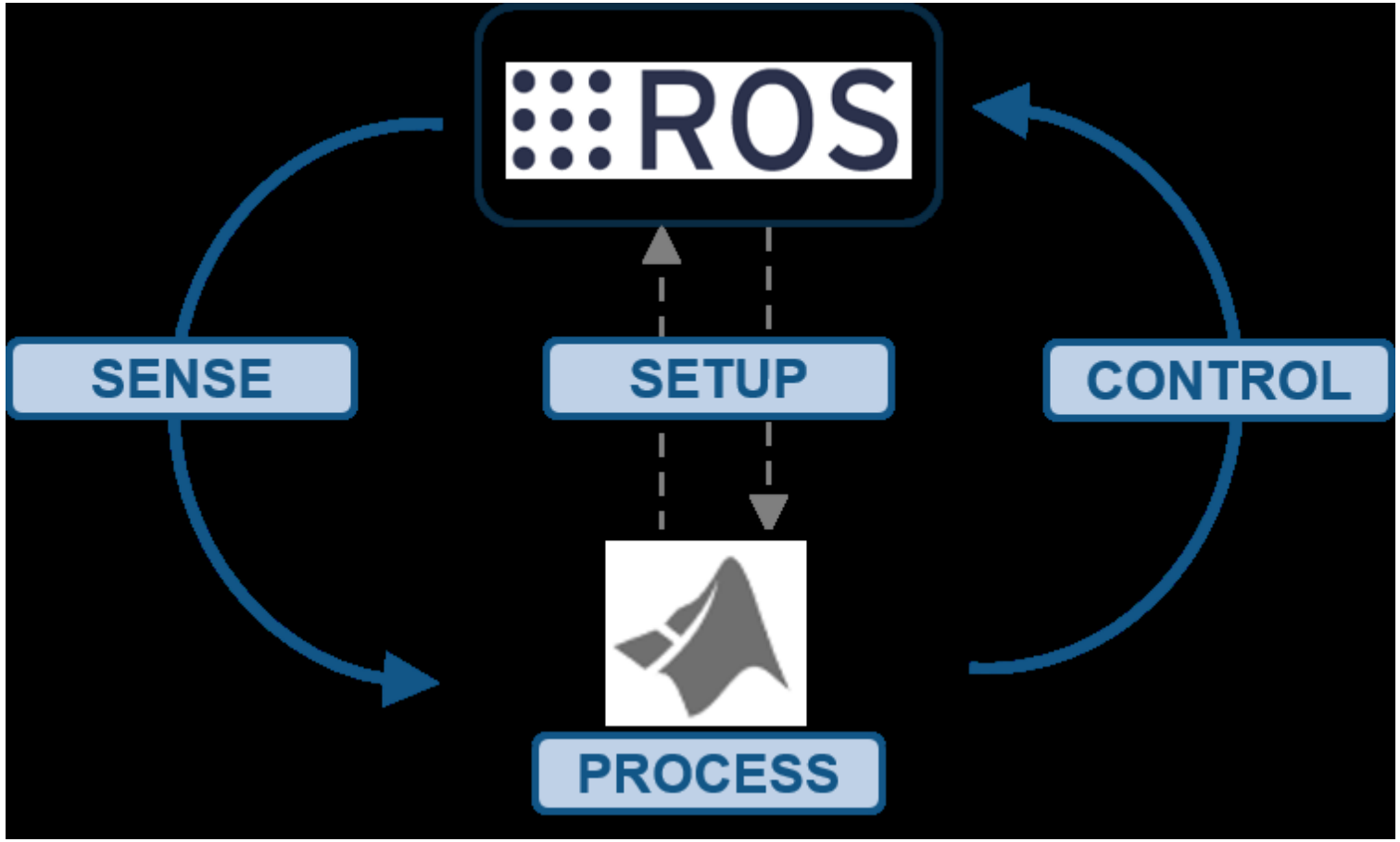
Khan Saad Bin Hasan
What is ROS?
Oct 20, 2019
https://towardsdatascience.com/what-why-and-how-of-ros-b2f5ea8be0f3

ROS, an open-source robot operating system. ROS is not an operating system in the traditional sense of process management and scheduling; rather, it provides a structured communications layer above the host operating systems of a heterogeneous compute cluster.[2]

Quigley, Morgan, et al. "ROS: an open-source Robot Operating System." ICRA workshop on open source software. Vol. 3. №3.2. 2009.

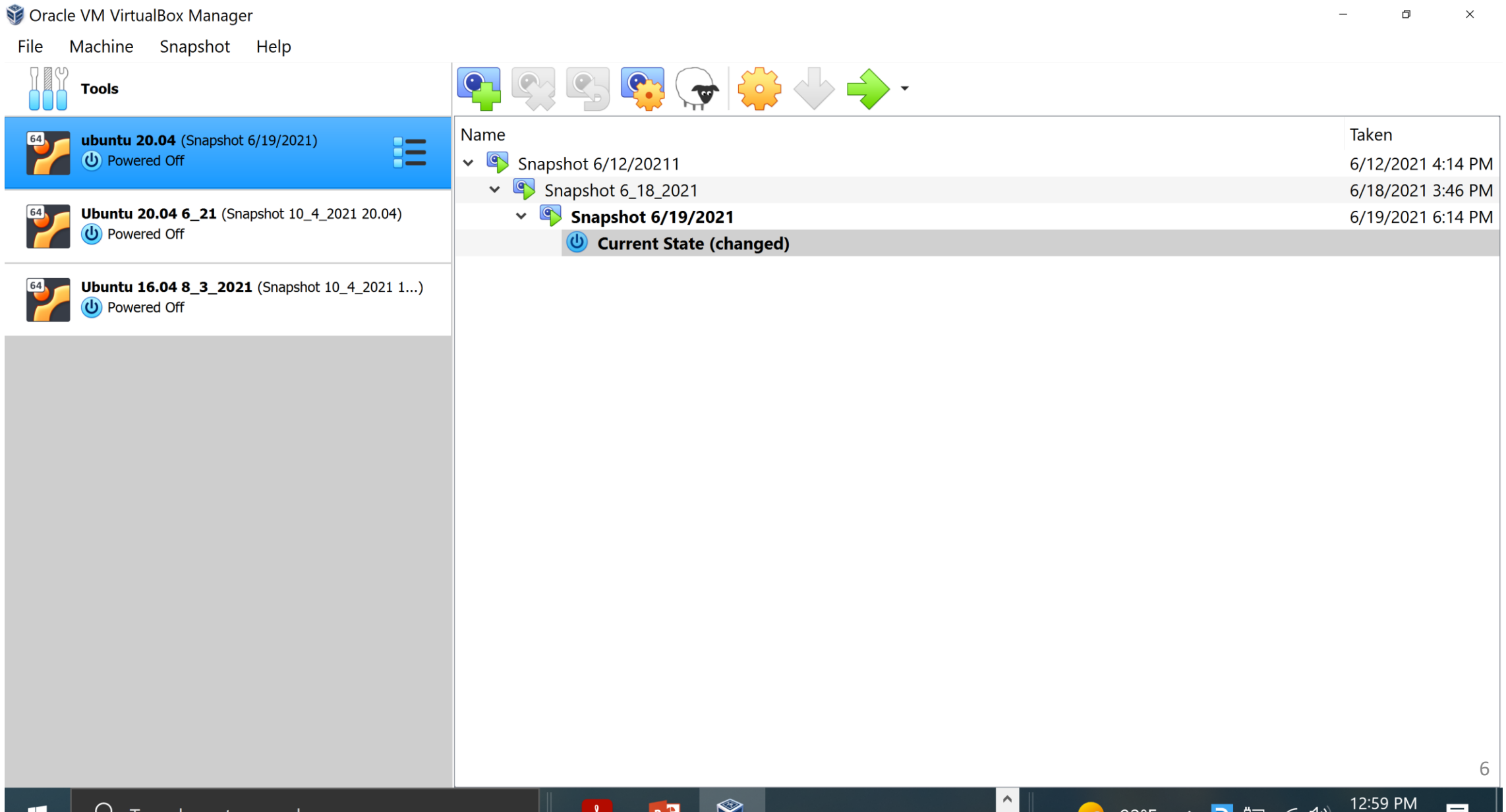


Robotic System

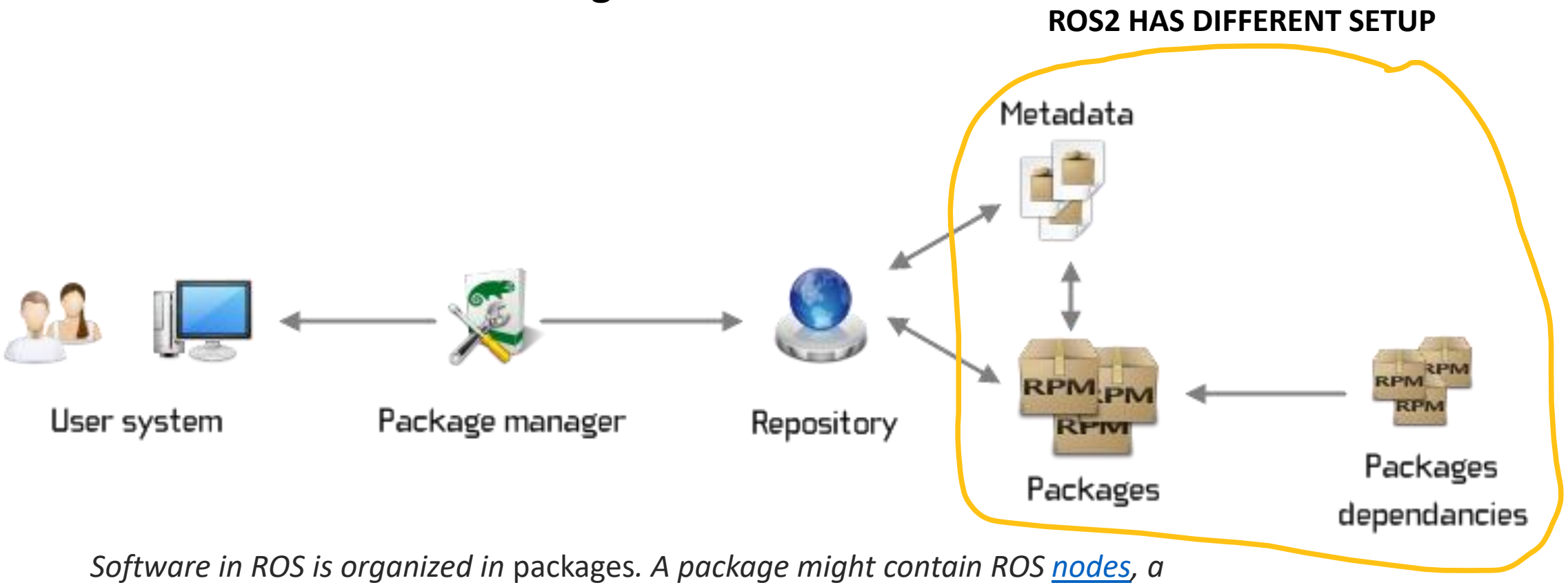


ROS is not an operating system but a meta operating system meaning, that it assumes there is an underlying operating system that will assist it in carrying out its tasks.

A GOOD APPROACH TO TEST DIFFERENT ROS DISTRIBUTIONS

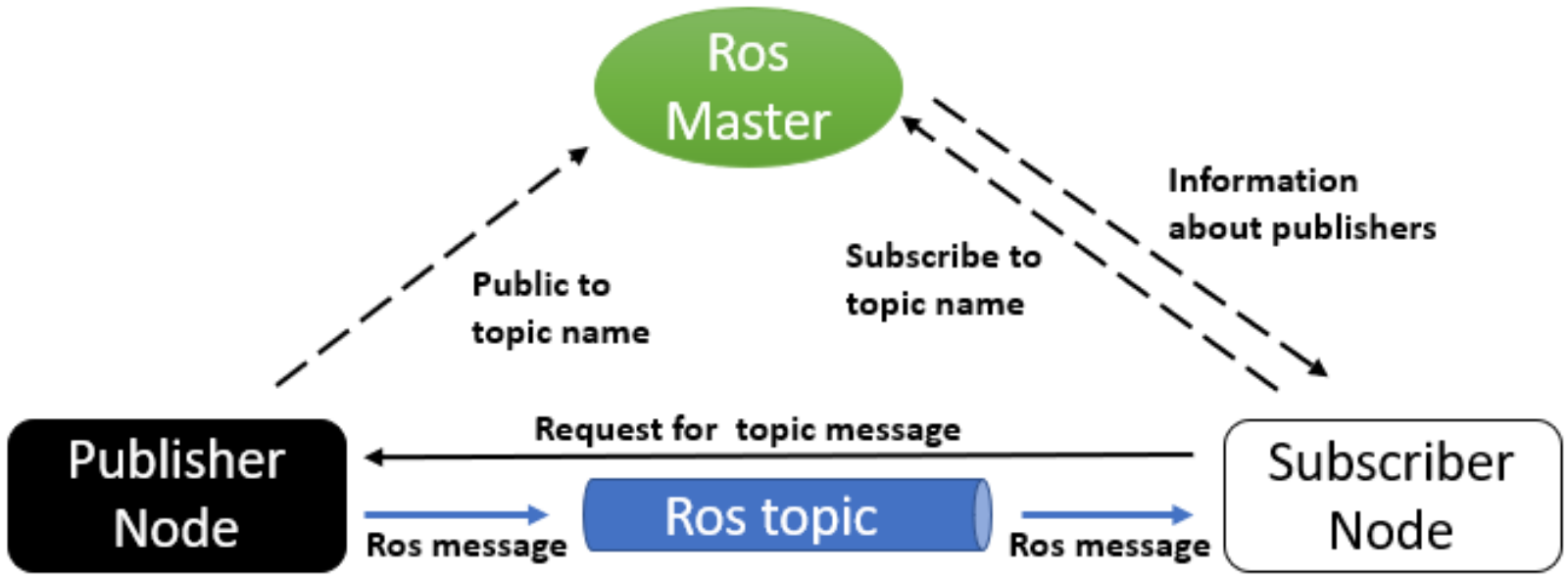


ROS 1 **Structure for Packages**



Software in ROS is organized in packages. *A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module.*

ROS 1 Structure



https://trojrobert.github.io/hands-on-introdution-to-robot-operating-system(ros)/

ROS2 Commands are a bit different – HOWEVER, ITEMS ARE BASCIALLY THE SAME

– BUT
NO ROSCORE.

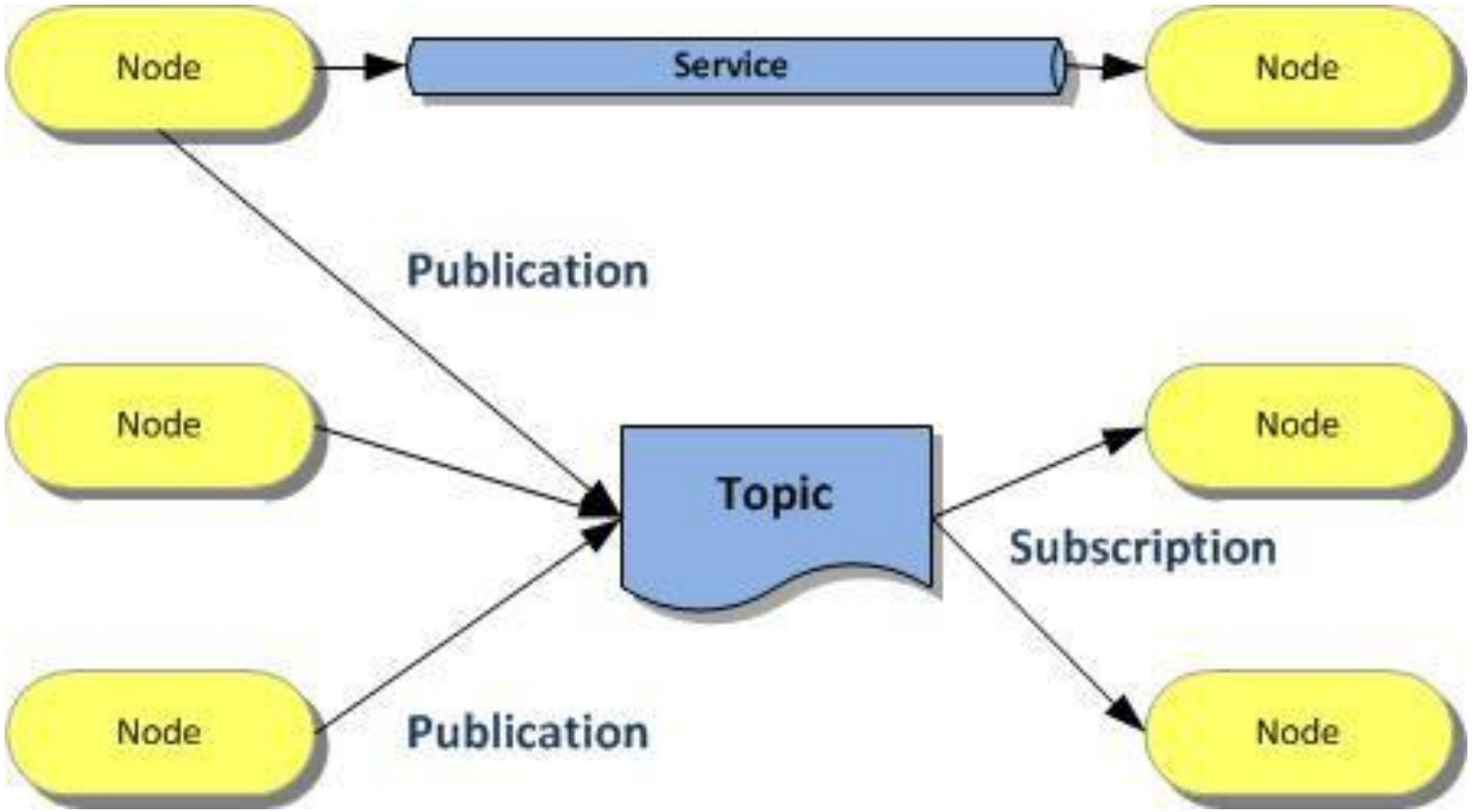
ACTIONS – are now a part of ROS2
– not just an addon

Command	Action	Example usage and subcommandexamples
roscore	Starts the Master	$ roscore
rosrun	Runs an executable programand creates nodes	$ rosrun [package name][executable name]

rosnode	Shows information about nodes and lists the activenodes	$ rosnode info [node name]
		$ rosnode<subcommand>
		Subcommand: list

rostopic	Shows information aboutROS topics	$ rostopic<subcommand><topicname> Subcommands: echo, info, and type
rosmsg	Shows information about themessage types	$ rosmsg<subcommand> [packagename]/ [message type] Subcommands: show, type, and list
rosservice	Displays the runtime information about variousservices and allows the display of messages beingsent to a topic	$ rosservice<subcommand>[service name] Subcommands: args, call, find,info, list, and type
rosparam	Used to get and set parameters (data) used bynodes	$ rosparam<subcommand>[parameter] Subcommands: get, set, list, and delete

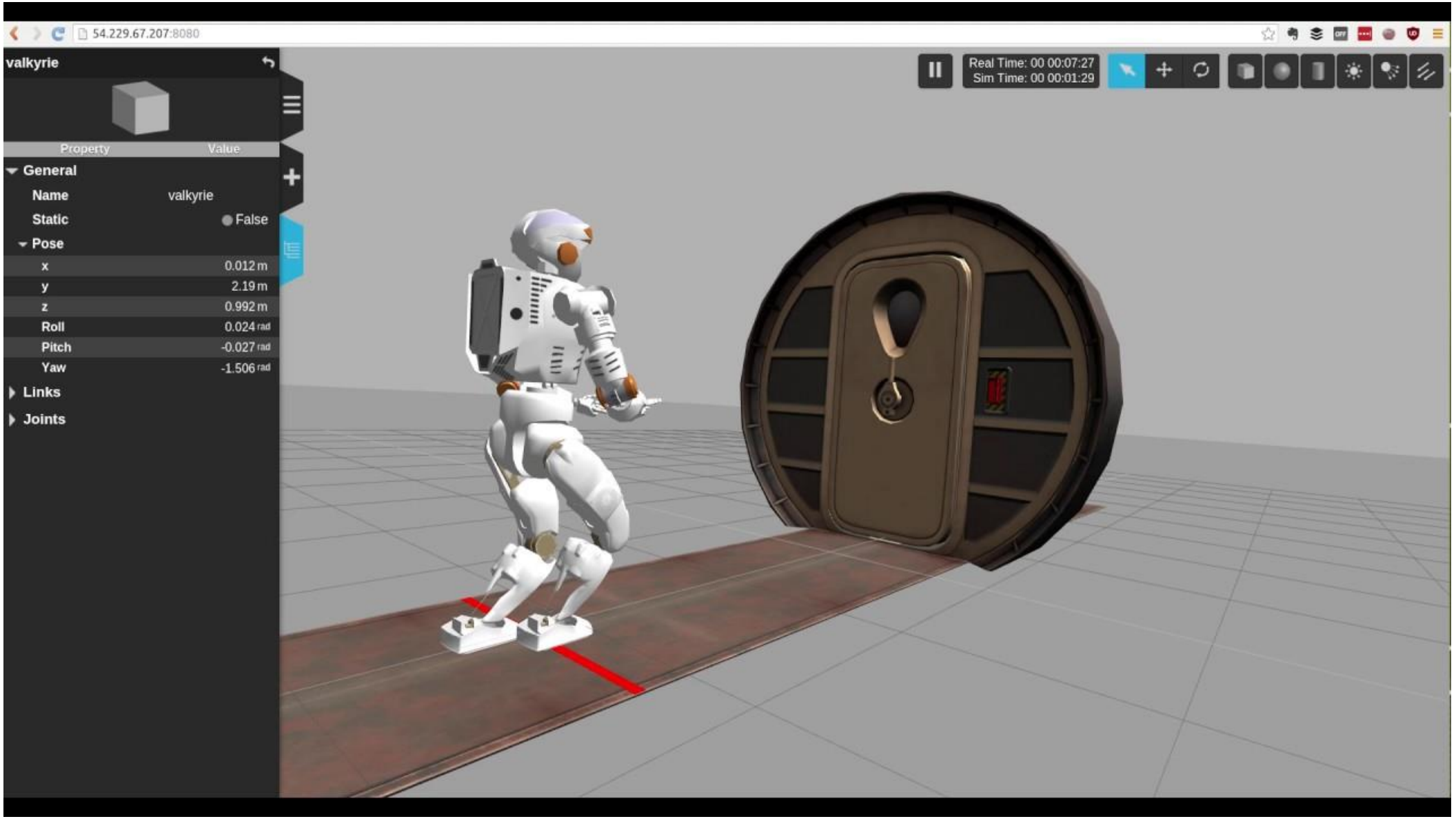
The website (http://wiki.ros.org/ROS/CommandLineTools) describes many ROS commands.


Node
Service
Node
Publication
Node
Topic
Node
Subscription
Node
Publication
Node

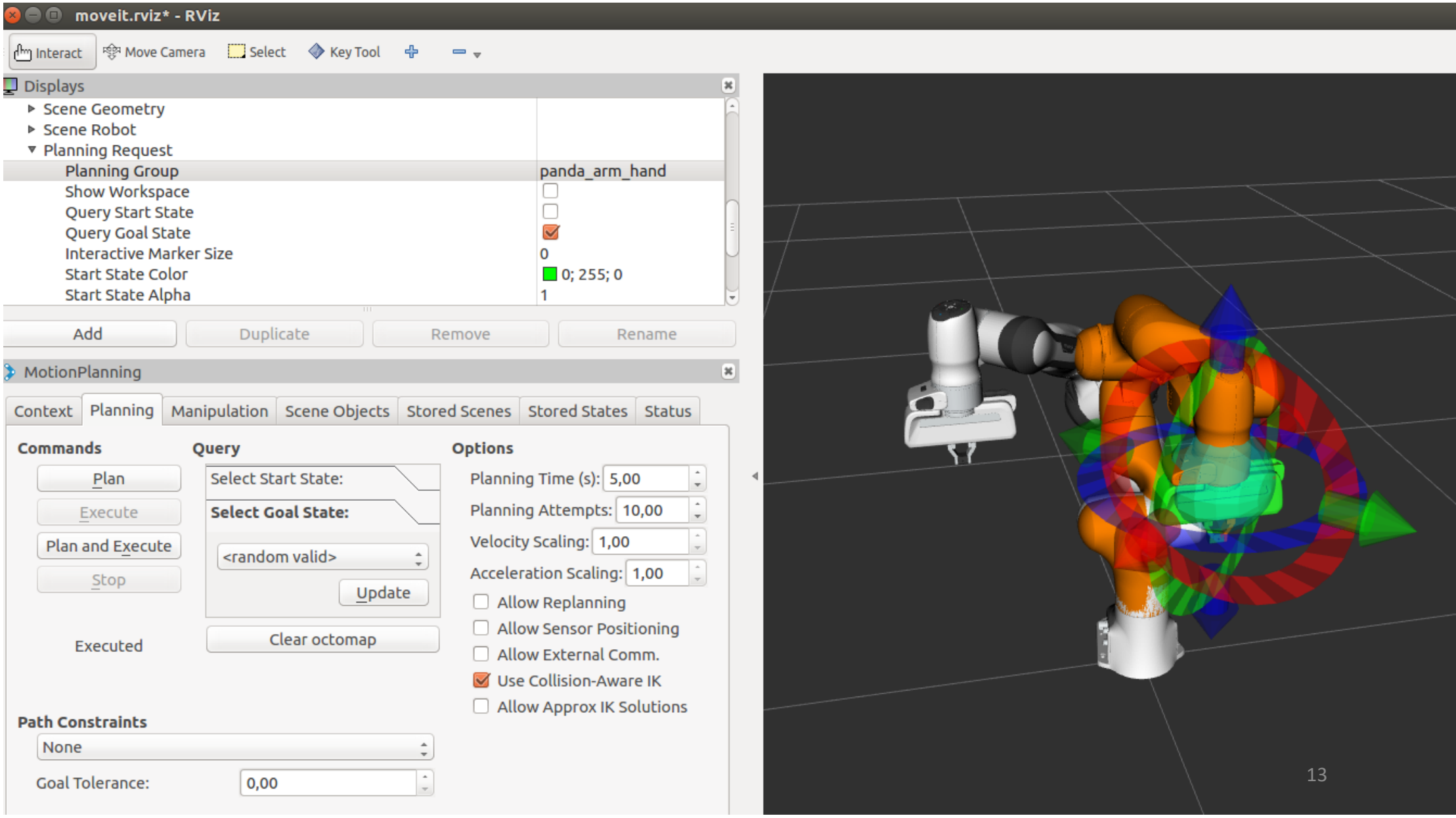
ROS 1 SUPPORT & SOFTWARE (UPDATED FOR ROS2)

- OPEN ROBOTICS
- CANONICAL (UBUNTU)
- GAZEBO RVIZ, OPENCV
- URDF, SDF (Gazebo), Xacro (XML macros)
- PYTHON, C AND C++ (Others for ROS1 – use with care)
- Various graphics and simulation packages
  - OGRE Graphics Rendering
  - ODE Open Dynamics engine
  - MoveIt
  - IK packages, SMACH

Tools: Gazebo Simulator



RVIZ Robot Visualizer MoveIt



Support Websites

- https://www.openrobotics.org/
- https://canonical.com/
- https://www.ogre3d.org/
- https://gazebosim.org/home
- https://www.ode.org/
- https://moveit.ros.org/
- http://wiki.ros.org/trac_ik Fast IK

Python 3 in Noetic

https://docs.python.org/3/howto/pyporting.html

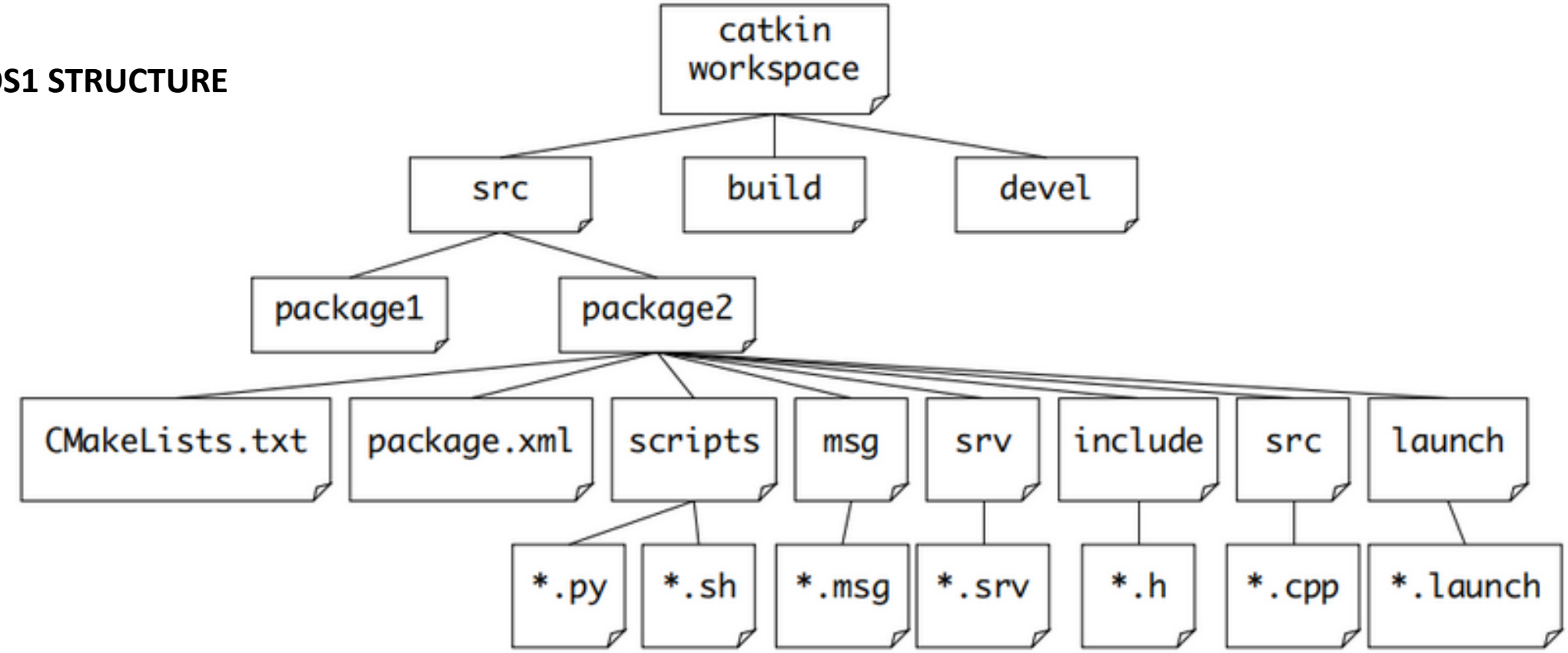
Gazebo 11 in Noetic

https://github.com/gazebosim/gazebo-classic/blob/master/Migration.md

https://github.com/gazebosim/gazebo-classic/blob/master/Changelog.md

ALWAYS CHECK THE DISTROS OF ROS AND UBUNTU - BUT ALSO THE COMPATIBLE SUPPORT SOFTWARE

ROS1 STRUCTURE



To compile our ROS1 workspace, use the *catkin_make* command to start the build process.

https://medium.com/swlh/7-simple-steps-to-create-and-build-our-first-ros-package-7e3080d36faa

catkin Build System

The catkin workspace contains the following spaces

Work here

src

The source space contains the source code. This is where you can clone, create, and edit source code for the packages you want to build.

Don't touch

build

The build space is where CMake is invoked to build the packages in the source space. Cache information and other intermediate files are kept here.

Don't touch

devel

The development (devel) space is where built targets are placed (prior to being installed).

Slide Credit: Marco Hutter, ETH Zurich

Some of the important files/directories inside Packages are:

1. Nodes: A node *is a process that performs computation.*
2. CMakeLists.txt: It is the input to the CMake build system for building software packages.
3. Package.xml : It defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages.
4. .yaml files: To run a rosnode you may require a lot of parameters e.g, Kp,Ki,Kd parameters in PID control. We can configure these using YAML files.
5. launch files: To run multiple nodes at once in ROS we use launch files.

Any code that will be written should be in the form of packages. And the packages should be inside a workspace*. Catkin is used in ROS1.

A catkin workspace is a folder where you modify, build, and install catkin packages. It can contain up to four different spaces *which each serve a different role in the software development process.*

1. The source space contains the source code of catkin packages. This is where you can extract/checkout/clone source code for the packages you want to build. Each folder within the source space contains one or more catkin packages.
2. The build space is where CMake is invoked to build the catkin packages in the source space. CMake and catkin keep their cache information and other intermediate files here.
3. The development space (or devel space) is where built targets are placed prior to being installed. The way targets are organized in the devel space is the same as their layout when they are installed. This provides a useful testing and development environment which does not require invoking the installation step.
4. Once targets are built, they can be installed into the install space by invoking the install target, usually with make install.

* **Python scripts with rospy can be run without being in a package.**

ROS WORKSPACE AND PACKAGE CREATION

WATCH THE VIDEO Noetic 6:19

https://www.youtube.com/watch?v=mwLsIhxUxQc

1,305 views Oct 18, 2022
Ubuntu Version : 20.04
ROS1 Version : NOETIC

http://wiki.ros.org/catkin/Tutorials/create_a_workspace

- source /opt/ros/noetic/setup.bash
- Make directories catkin_ws and src (Catkin name is arbitrary)
- $ source ~/catkin_ws/devel/setup.bash and $ echo $ROS_PACKAGE_PATH – See ws and ros

http://wiki.ros.org/ROS/Tutorials/CreatingPackage

http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

Let's Go Through These tutorials - Link of ROSPY AND PYTHON

1. Source the ROS Distribution

Alias foxy or noetic

harman@harman-VirtualBox:~$ **noetic** **(This sources noetic via an alias)** **(not foxy)**

harman@harman-VirtualBox:~$ **gedit .bashrc** **(bashrc is hidden!)**

…

#source /opt/ros/foxy/setup.bash # 6_21_2021 Load Foxy

echo Alias foxy or noetic

alias foxy='source /opt/ros/foxy/setup.bash' # Load Foxy,7_30_2021 or noetic

alias noetic='source /opt/ros/noetic/setup.bash

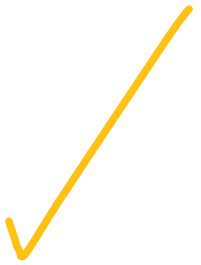
2. SOURCE THE WORKSPACE TO Execute Code - ros_robotics

harman@harman-VirtualBox:~$ **source ~/catkin_ws/devel/setup.bash**

3. Now Check the paths

harman@harman-VirtualBox:~$ **env | grep ROS_PACKAGE_PATH**

ROS_PACKAGE_PATH=/home/harman/catkin_ws/src:/opt/ros/noetic/share



How Does rosrun work? - $ roscore running

- harman@harman-VirtualBox:~$ **noetic**
- ROS_DISTRO was set to 'foxy' before. Please make sure that the environment does not mix paths from different distributions.
- harman@harman-VirtualBox:~$ **rosrun turtlesim turtlesim_node**
- [INFO] [1668973275.885955332]: Starting turtlesim with node name /turtlesim
- [INFO] [1668973275.891606121]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]

https://github.com/ros/ros/blob/0cf372d5225045ecae083ce210e0f1a2cbe6f8b8/tools/rosbash/scripts/rosrun

VIEW CODE ON GITHUB

```
#!/usr/bin/env bash

function usage() {
  echo "Usage: rosrun [--prefix cmd] [--debug] PACKAGE EXECUTABLE [ARGS]"
  echo "  rosrun will locate PACKAGE and try to find"
  echo "  an executable named EXECUTABLE in the PACKAGE tree."
  echo "  If it finds it, it will run it with ARGS."
}
```

catkin_package_libexec_dirs=($(catkin_find --without-underlays --libexec --share "$pkg_name" 2> /dev/null))

https://www.theconstructsim.com/ros-5-mins-007-rosrun-works/ Short Video

USING A LAUNCH FILE

- Starts roscore
- Launch multiple nodes
- Sets parameters on the parameter server

http://wiki.ros.org/roslaunch

http://wiki.ros.org/roslaunch/XML/node

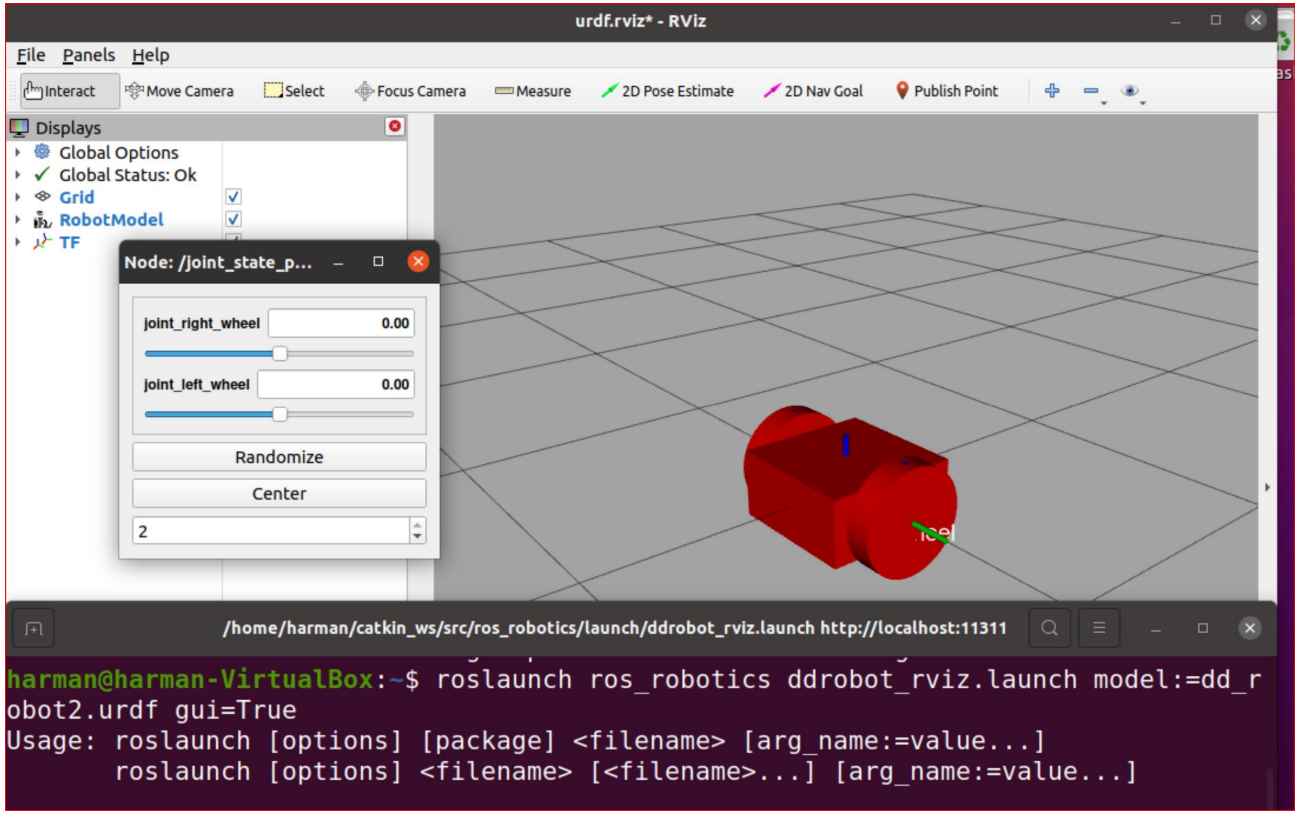
https://wiki.ros.org/Parameter%20Server

Example package Chapter 2 Package Directory for ros_robotics

```
harman@harman-VirtualBox:~$ cd ~/catkin_ws/src
harman@harman-VirtualBox:~/catkin_ws/src$ ls
 CMakeLists.txt  ros_robotics
harman@harman-VirtualBox:~/catkin_ws/src$ cd ros_robotics
harman@harman-VirtualBox:~/catkin_ws/src/ros_robotics$ tree -L 1
.
```

- **CMakeLists.txt**
- **launch**
- **package.xml**
- **urdf**
- **urdf.rviz**
- **worlds**

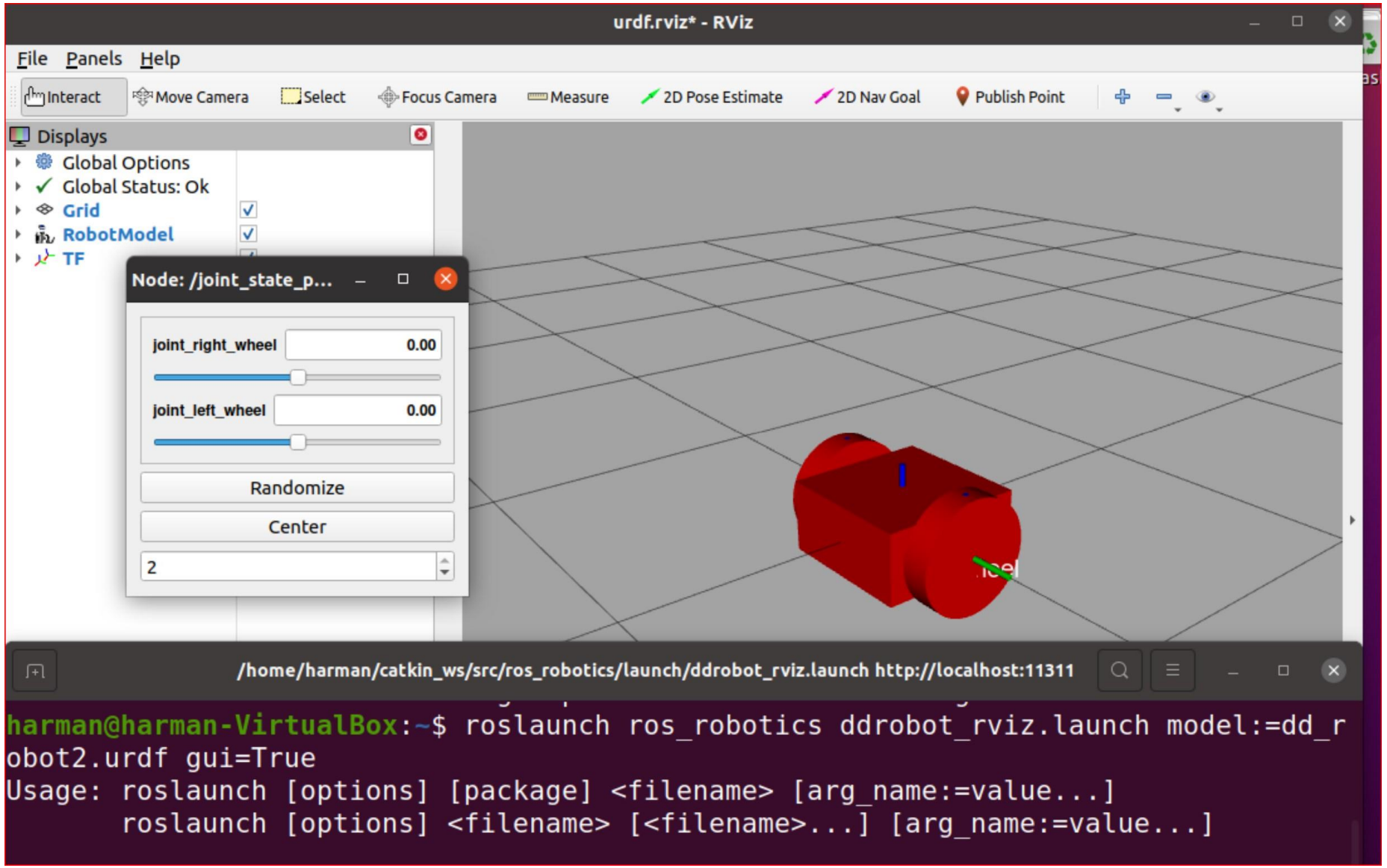
3 directories, 3 files



Note – Addition of Launch File

```
<launch>
  <!-- values passed by command line input The model i.e. dd_robotx.urdf-->
  <arg name="model" />
    <!-- <arg name="gui" default="False" /> 7/30/21 Put gui:=True on Command Line-->

  <!-- set these parameters on Parameter Server -->
  <param name="robot_description" textfile="$(find ros_robotics)/urdf/$(arg model)" />
  <param name="use_gui" value="$(arg gui)"/>


  <!-- Start 3 nodes: joint_state_publisher_gui, robot_state_publisher and rviz -->
  <node name="joint_state_publisher_gui" pkg="joint_state_publisher_gui" type="joint_state_publisher_gui" />

  <node name="robot_state_publisher" pkg="robot_state_publisher"
   type="robot_state_publisher" />
  <!-- state_publisher changed to robot_state_publisher -->

  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find ros_robotics)/urdf.rviz" required="true" />
  <!-- (required = "true") if rviz dies, entire roslaunch will be killed -->
</launch>
```

Launch file was modified from ROS Kinetic to ROS noetic. Note gui=True



Joint State Publisher GUI Migration to Noetic

In previous versions of ROS, the joint_state_publisher package had a parameter called use_gui that would launch a GUI when joint_state_publisher was started.

In early 2020 this package was split into a joint_state_publisher and joint_state_publisher_gui package. In Noetic, the use_gui parameter has been removed completely, and instead users should explicitly invoke joint_state_publisher_gui when they wish to use the GUI.

http://wiki.ros.org/noetic/Migration

CLIENT LIBRARIES

PYTHON AND C

- Python is VERY sensitive to spacing - normally indent 4 spaces
- **When copying code from a file IF an error –**
- SyntaxError: invalid character in identifier
- IndentationError: expected an indented block
- **RETYPE THE LINE AND WATCH SPACING**

I. TALK ABOUT ROSPY AND RCLPY Python

API

An API, or Application Programming Interface, is an interface that is provided by an “application”, which in this case is usually a shared library or other language appropriate shared resource. APIs are made up of files that define a contract between the software using the interface and the software providing the interface. These files typically manifest as **header files in C and C++ and as Python files in Python**. In either case it is important that APIs are grouped and described in documentation and that they are declared as either public or private. Public interfaces are subject to change rules and changes to the public interfaces prompt a new version number of the software that provides them.

client_library

A client library is an API that provides access to the ROS graph using primitive middleware concepts like Topics, Services, and Actions.

ROS Client Libraries

Client Library	Language	Comments
roscpp	C++	Most widely used, high performance
rospy	Python	Good for rapid-prototyping and non-critical-path code
roslisp	LISP	Used for planning libraries
rosjava	Java	Android support
roslua	Lua	Light-weight scripting
roscs	Mono/.Net	Any Mono/.Net language
roseus	EusLisp	
PhaROS	Pharo Smalltalk	
rosR	R	Statistical programming

Experimental

Client API Commonly Used Features

Object / Feature	Description	roscpp	rospy
API root	Objects and methods for interacting with ROS	ros::NodeHandle	rospy
Parameter server client	Query and set parameter server dictionary entries	.getParam .param .searchParam .setParam	.get_param .search_param .set_param
Subscriber	Receive messages from a topic	.subscribe	.Subscriber
Publisher	Send messages to a topic	.advertise	.Publisher
Service	Serve and call remote procedures	.advertiseService .serviceClient	.Service .ServiceProxy
Timer	Periodic interrupt	.createTimer	.Timer
Logging	Output strings to rosconsole	ROS_DEBUG, ROS_INFO, ROS_WARN, etc.	.logdebug, .loginfo, .logwarn, .logerr, .logfatal
Initialization & Event Loop	Set node name, contact Master, enter main event loop	ros::init .spin	.init_node .spin
Messages	Create and extract data from ROS messages	Specifics depends on message	
		std_msgs::String	std_msgs.msg.String

rospy client library: Example

```
1 import rospy
2 from std_msgs.msg import String
3
4 pub = rospy.Publisher('topic_name', String, queue_size=10)
5 rospy.init_node('node_name')
6 r = rospy.Rate(10) # 10hz
7 while not rospy.is_shutdown():
8     pub.publish("hello world")
9     r.sleep()
```

rospy client library: Initializing your ROS Node

```
rospy.init_node('my_node_name')
```

and

```
rospy.init_node('my_node_name', anonymous=True)
```

You can only have one node in a rospy process,

so you can only
call rospy.init_node()once.

Names have important properties in ROS.
Most importantly, they must be **unique**.
In cases where you don't care about unique names for a particular node, you may wish to initialize the node with an *anonymous* name.

rospy client library: Testing for shutdown

```
while not rospy.is_shutdown():
    do some work
```

and

```
... setup callbacks
rospy.spin()
```

The spin() code simply sleeps until the is_shutdown() flag is True.

There are multiple ways in which a node can receive a shutdown request, so it is important that you use one of the two methods above for ensuring your program terminates properly.

rospy client library: Message generation

- package_name/msg/Foo.msg → package_name.msg.Foo
- rospy takes msg files and generates Python source code for them.

n your code you would use
:

```
import std_msgs.msg
msg = std_msgs.msg.String()
```

or

```
from std_msgs.msg import String
msg = String()
```

rospy client library: std_msgs

std_msgs/String Message

File: std_msgs/String.msg

Raw Message Definition

```
string data
```

Compact Message Definition

```
string data
```

2. ROS Message Types

ROS Message Types

- Bool
- Byte
- ByteMultiArray
- Char
- ColorRGBA
- Duration
- Empty
- Float32
- Float32MultiArray
- Float64
- Float64MultiArray
- Header
- Int16
- Int16MultiArray
- Int32
- Int32MultiArray
- Int64
- Int64MultiArray
- Int8
- Int8MultiArray
- MultiArrayDimension
- MultiArrayLayout
- String
- Time
- UInt16
- UInt16MultiArray
- UInt32
- UInt32MultiArray
- UInt64
- UInt64MultiArray
- UInt8
- UInt8MultiArray

rospy client library: Publishing to a topic

Create a handle to publish messages to a topic using the rospy.Publisher class

```
pub = rospy.Publisher('topic_name', std_msgs.msg.String, queue_size=10)
pub.publish(std_msgs.msg.String("foo"))
```

You can then call publish() on that handle to publish a message

Toggle line numbers

```
1  #!/usr/bin/env python
2  # license removed for brevity
3  import rospy
4  from std_msgs.msg import String
5
6  def talker():
7      pub = rospy.Publisher('chatter', String, queue_size=10)
8      rospy.init_node('talker', anonymous=True)
9      rate = rospy.Rate(10) # 10hz
10     while not rospy.is_shutdown():
11         hello_str = "hello world %s" % rospy.get_time()
12         rospy.loginfo(hello_str)
13         pub.publish(hello_str)
14         rate.sleep()
15
16 if __name__ == '__main__':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
20         pass
```

makes sure your script is executed as a Python script

The std_msgs.msg import is so that we can reuse the std_msgs/String message type

publishing to the chatter topic using the message type String

tells rospy the name of your node

creates a Rate object rate.

checking the rospy.is_shutdown() flag

Create the message

the messages get printed to screen, it gets written to the Node's log file, and it gets written to rosout

publishes a string to our chatter topic

sleeps just long enough to maintain the desired rate through the loop.

Python __main__ check

This catches a rospy.ROSInterruptException exception, which can be thrown by rospy.sleep() and rospy.Rate.sleep() methods when Ctrl-C is pressed

```
#!/usr/bin/env python              # The first line makes sure your script is executed as a Python script (python3).
 import rospy

from std_msgs.msg import String     # The output will be a string

def talker():

     pub = rospy.Publisher('chatter', String, queue_size=10)   # Chatter Topic

      rospy.init_node('talker', anonymous=True)     # This tells rospy the name of your node
      rate  =rospy.Rate(10)    # 10hz


     while not rospy.is_shutdown():
          hello_str = "hello world %s" % rospy.get_time()

          rospy.loginfo(hello_str)    # printed to screen, written to the Node's log file, and written to rosout

          pub.publish(hello_str)      #

          rate.sleep()

if __name__ == '__main__':

   try:

      talker()

   except rospy.ROSInterruptException:     # CNTL + C to end

       pass
```

RUN ROSCORE
Alias foxy or noetic
harman@harman-VirtualBox:~$ **noetic**
harman@harman-VirtualBox:~$ **roscore**

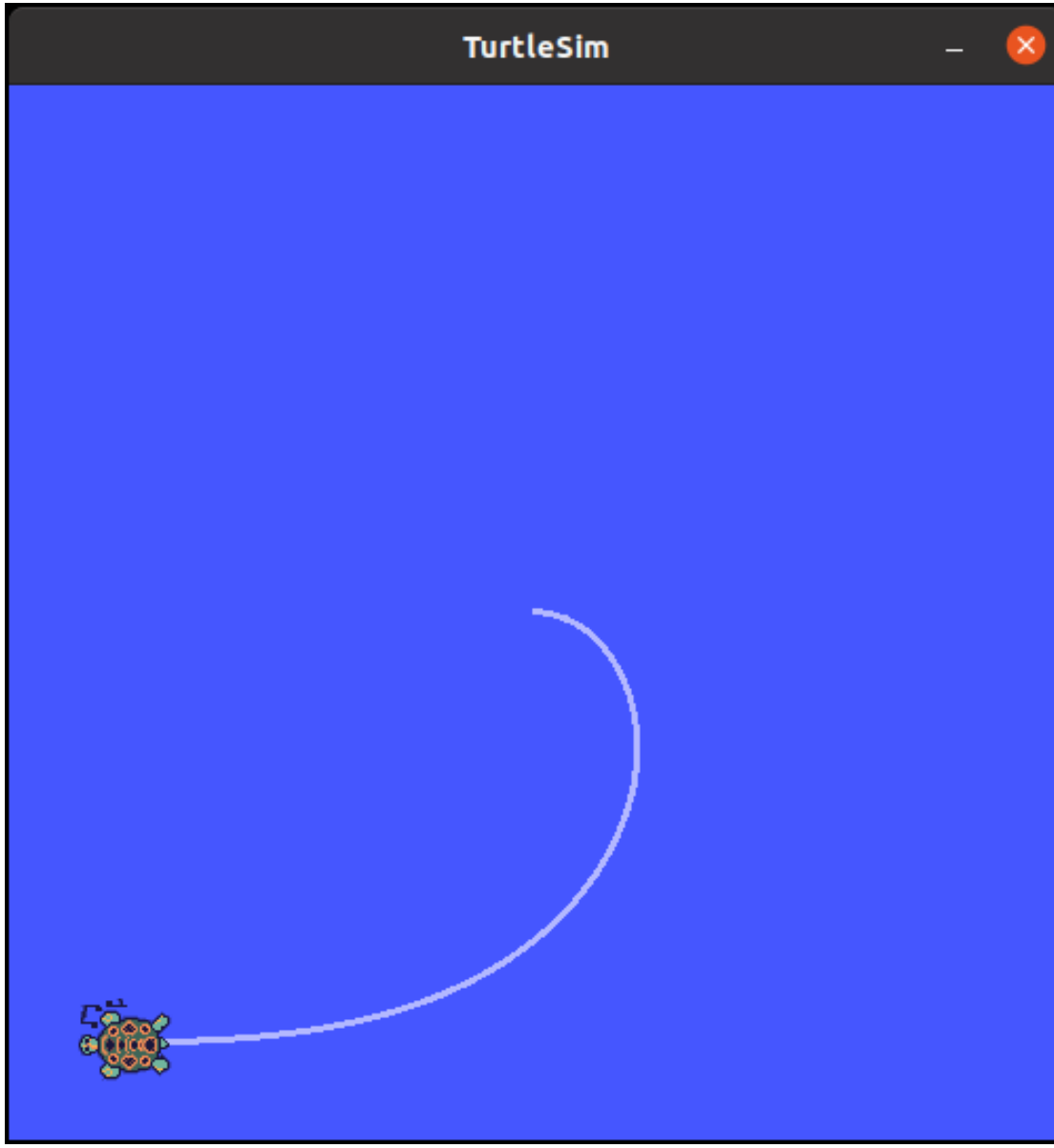
TERMINAL 2

harman@harman-VirtualBox:~/Desktop$ python3 publishHello.py

harman@harman-VirtualBox:~/Desktop$ python3 publishHello.py
[INFO] [1668979420.500695]: hello world 1668979420.5006263
[INFO] [1668979420.603913]: hello world 1668979420.603789
[INFO] [1668979420.701357]: hello world 1668979420.7012498
[INFO] [1668979420.804461]: hello world 1668979420.804322
[INFO] [1668979420.900951]: hello world 1668979420.9008462
[INFO] [1668979421.001503]: hello world 1668979421.0013928
[INFO] [1668979421.10

turtlesim_py_1 gotogoal 10_13_2022.pdf

(Go over this code)



Turtlesim Noetic gotogoal_1 10_13_2022

gotogoal_1 Python3 P control 10_13_2022 Corrected

#!/usr/bin/env python gotogoal_1.py Python3
turtlesim_cleaner/src/gotogoal.py GitHub

 # Added float - float(input("Set your x goal:"))

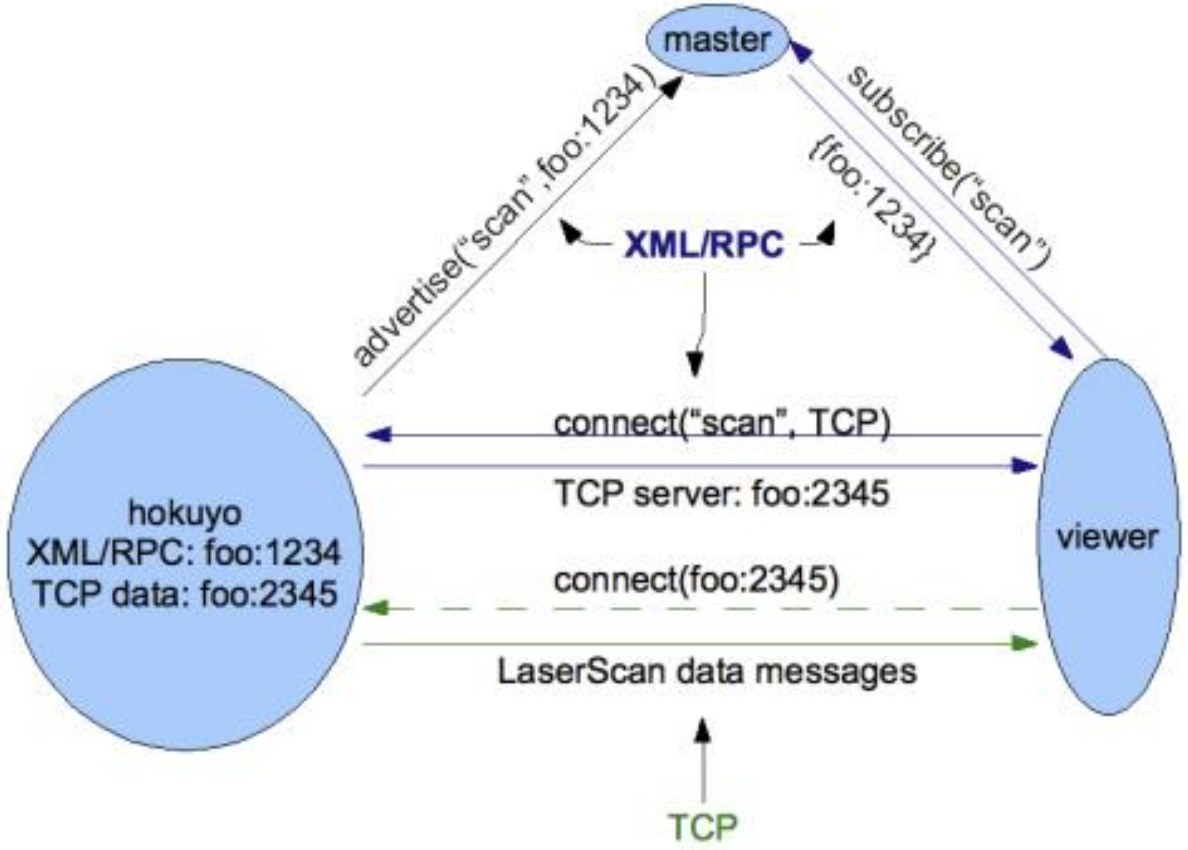
```
import rospy
from geometry_msgs.msg  import Twist
from turtlesim.msg import Pose
from math import pow,atan2,sqrt
```

OOP Object Oriented

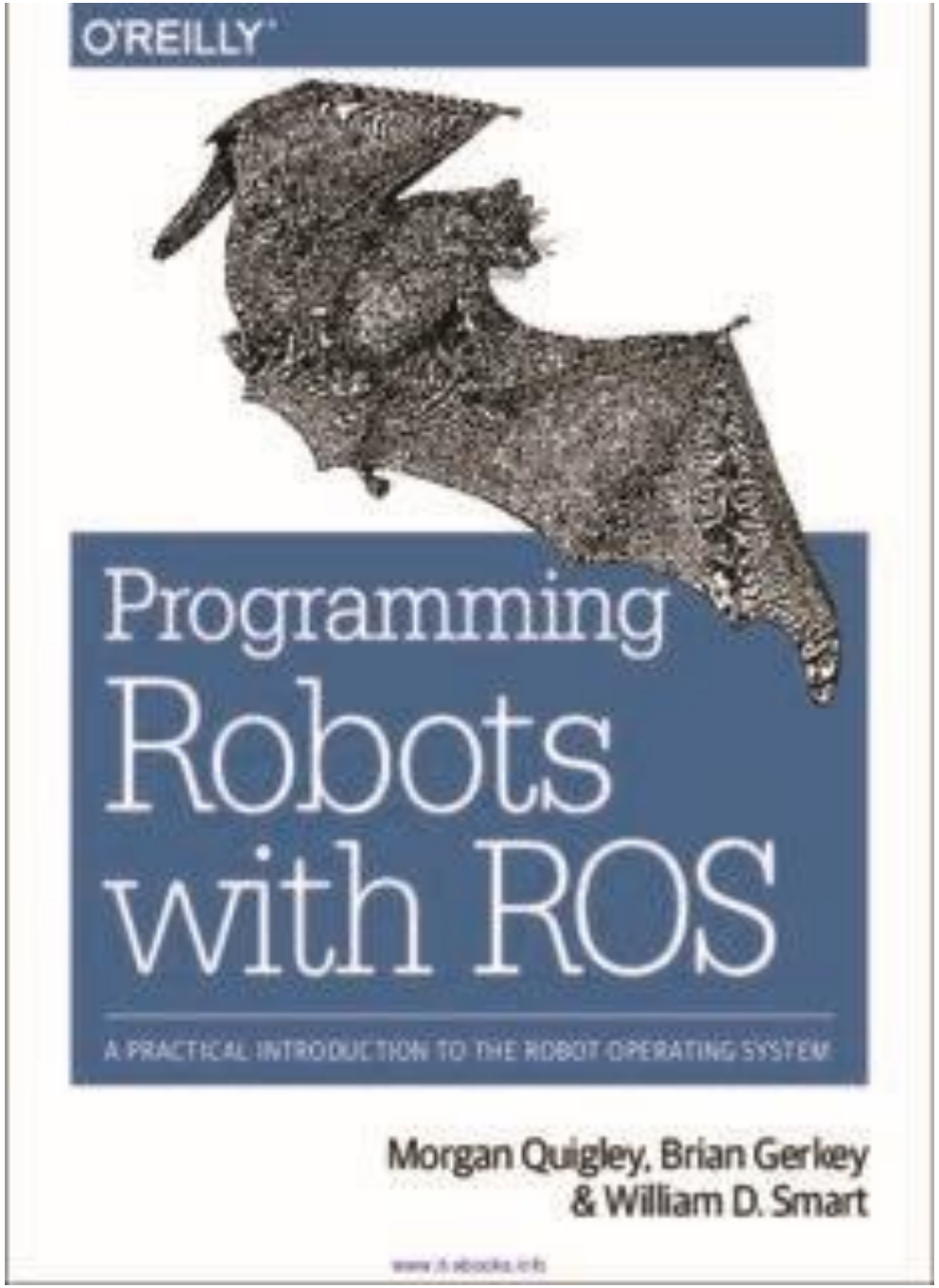
class turtlebot():

```
def __init__(self):
    #Creating our node,publisher and subscriber
    rospy.init_node('turtlebot_controller', anonymous=True)
    self.velocity_publisher = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)
    self.pose_subscriber = rospy.Subscriber('/turtle1/pose', Pose, self.callback)
    self.pose = Pose()
    self.rate = rospy.Rate(10)
```

ROS1 COMMUNICATION



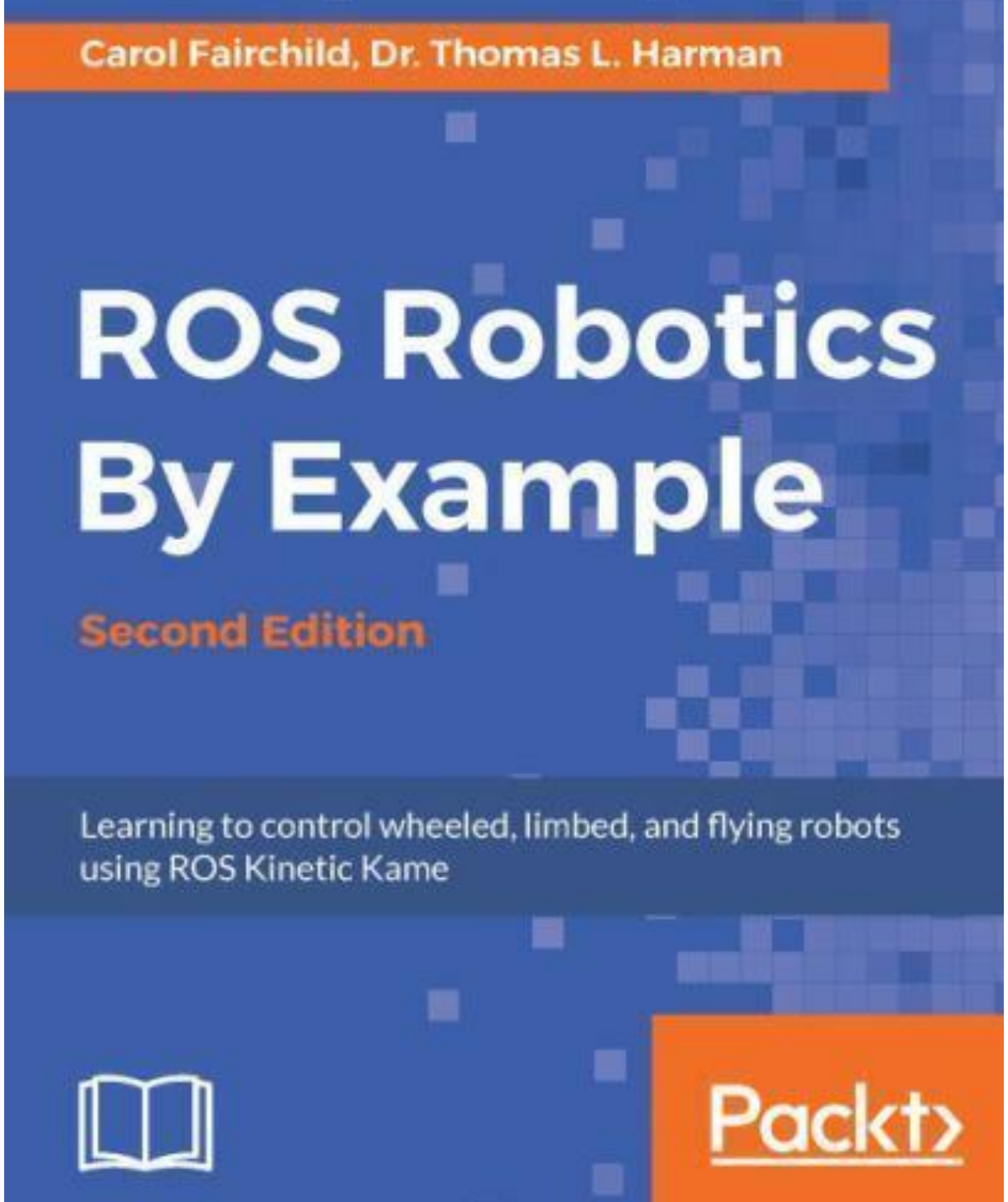
http://wiki.ros.org/ROS/Technical%20Overview



447 Pages · 2015 · 32.43 MB · 18,725 Downloads· English

INDIGO

From the Horses' Mouths



Kinetic Version