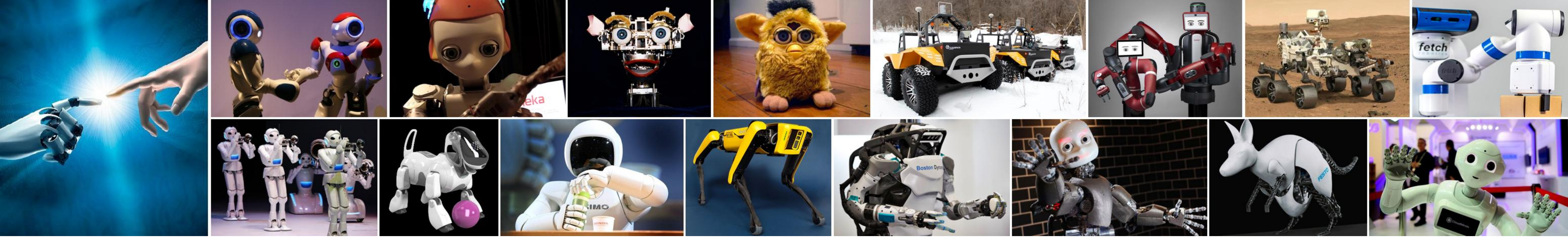
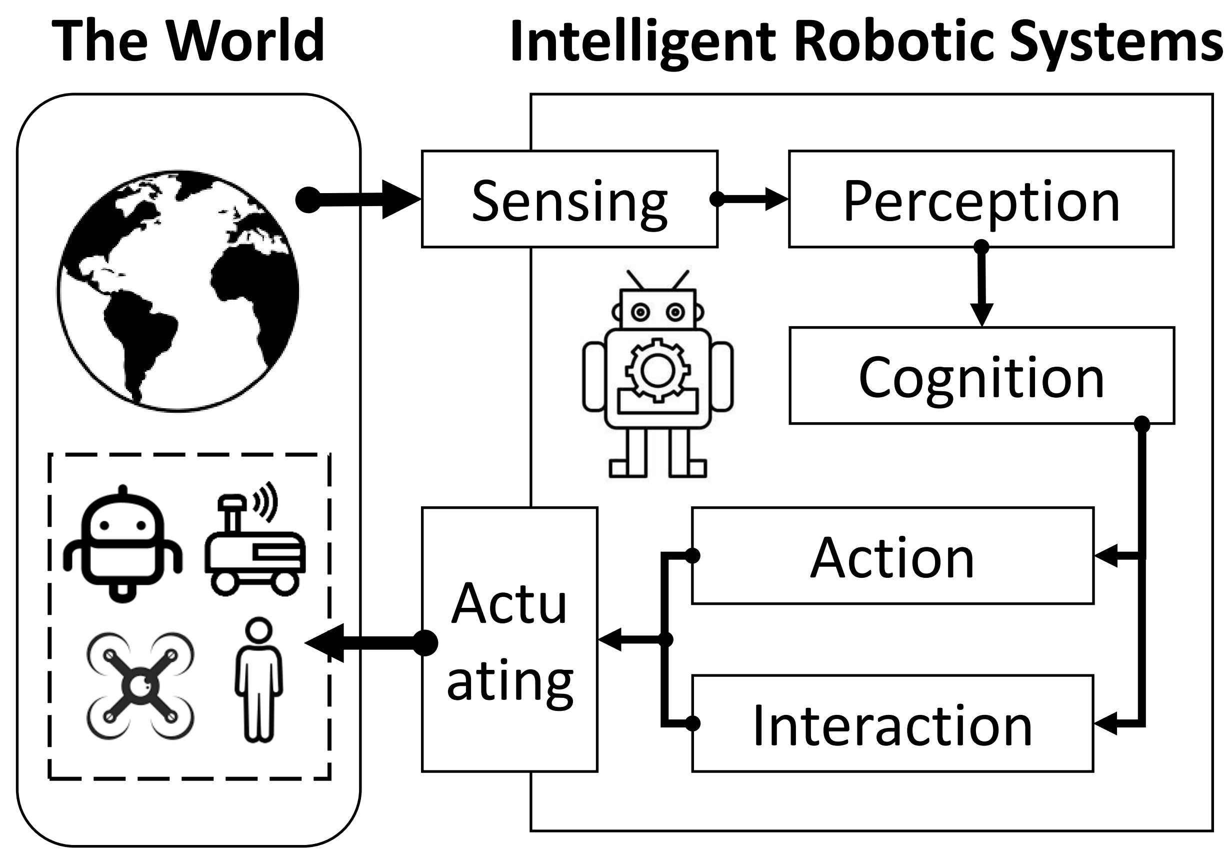


COMPSCI-603: Robotics

Software for AI Robots

Let's design a robot together



Let’s design a robot together

- Sensors:
  - RGBD camera: Intel RealSense D435
  - 360-degree LiDAR: RPLIDAR
  - Wheel encoder
- Embedded computers:
  - Nvidia Jetson Nano
  - Arduino
- Actuators: 63:1 metal gearmotor x 3
- Interaction:
  - LED ring
  - Controller

Hardware/Software Components

Path Planning

Safety Monitoring

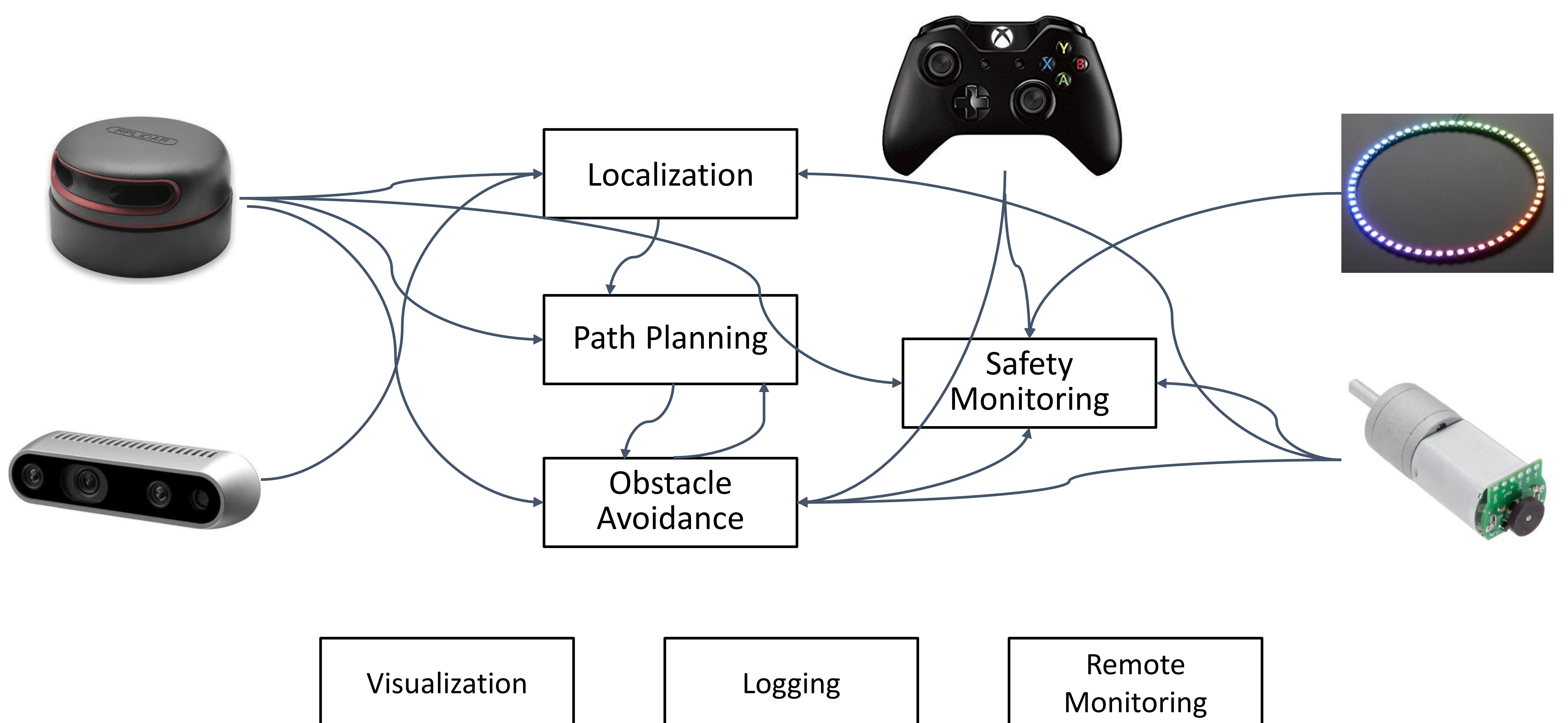
Obstacle Avoidance

Visualization

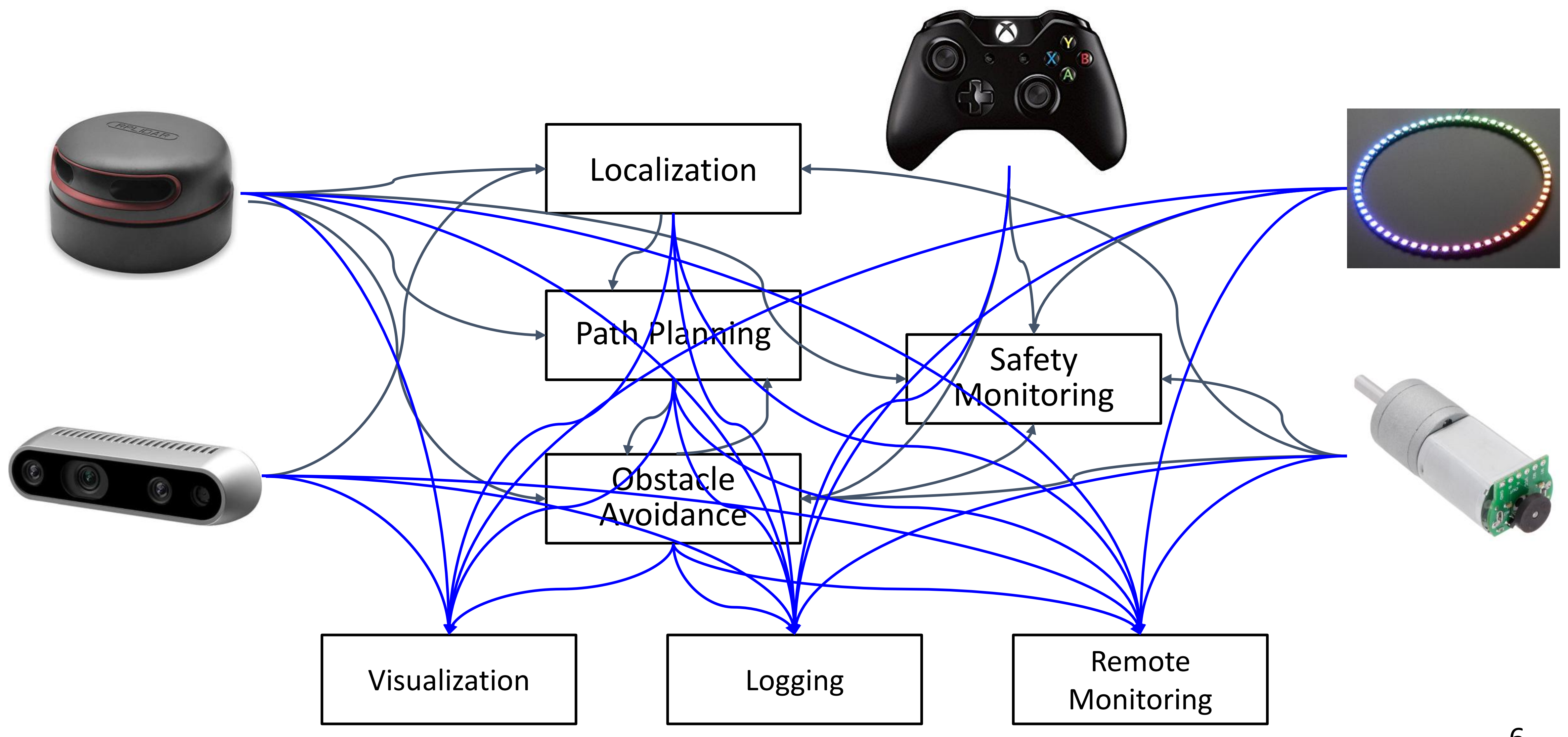
Logging

Remote Monitoring

Hardware/Software Components



Hardware/Software Components



Hardware/Software Components

Besides, an AI robot may need: object detection, facial recognition, manipulation, language-based interaction, …

And…

What if we want to design two *different* robots?

What a headache!

How can we deal with these software components?

ROS: Robot Operating System

ROS is open-source middleware that provides software libraries and tools for robotics development

ROS: Robot Operating System

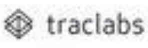
- ROS has two “sides”:
  - The operating system side, which provides standard operating system services such as:
    - hardware abstraction
    - low-level device control
    - implementation of commonly used functionality
    - message-passing between processes
    - package management
  - A suite of user contributed packages that implement common robot functionality such as perception, mapping and localization, planning, manipulation, etc.

ROS: Robot Operating System

- Modular design.
- Reusable robotics components.
- Hundreds of robots supported https://robots.ros.org.
- Now: (unofficially) almost all open-source robots are using certain components of ROS…
- Hundreds of ready to use algorithms.
- Efficient, so it can be used for actual products, not just prototyping.
- ROS1 runs on Ubuntu, and ROS2 supports multiple OS.
- Parallelization and networking made easy, can use multiple machines simultaneously.

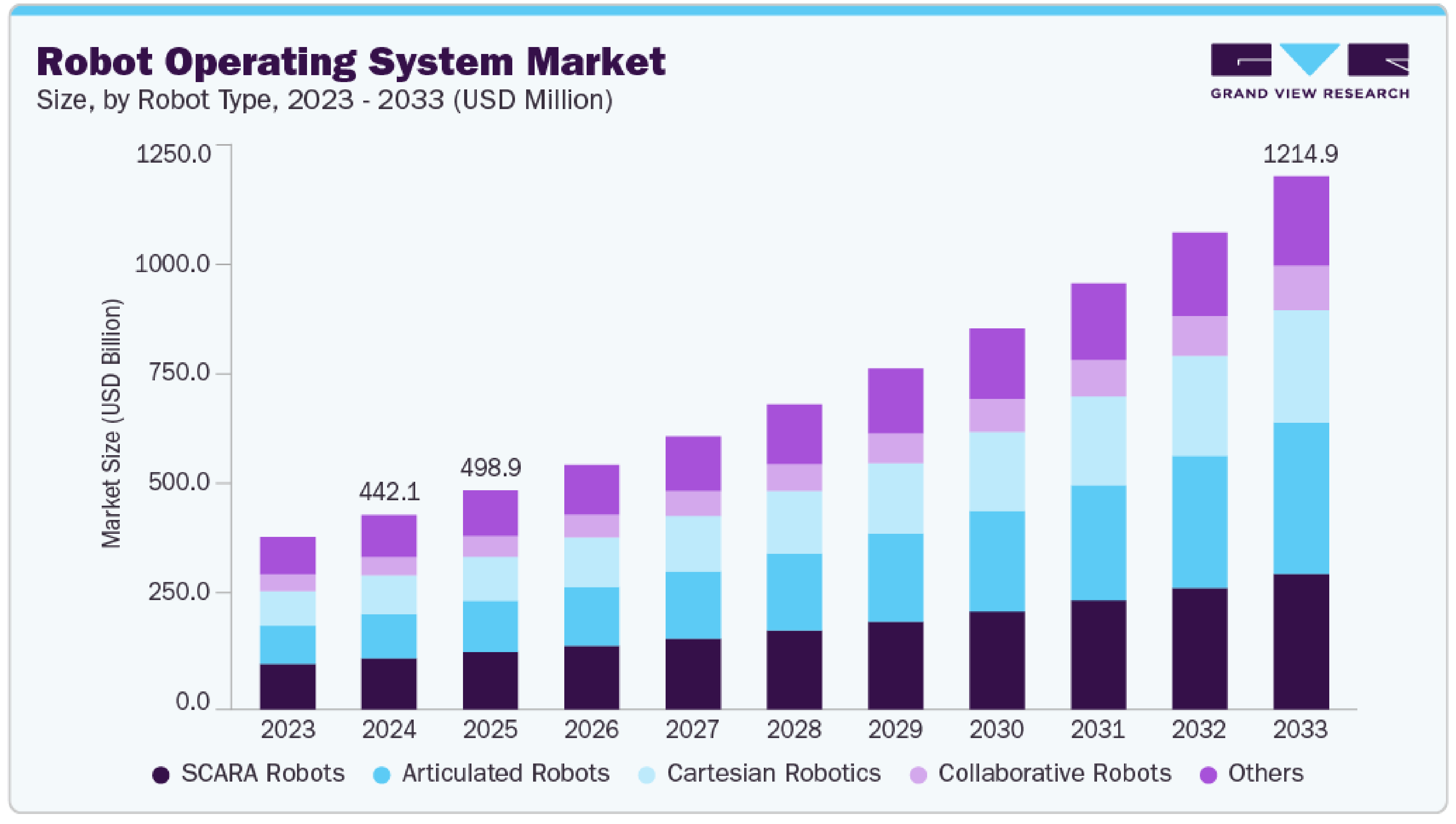
ROS: Robot Operating System

- Widely used in robotics industry jobs.



https://rosindustrial.org/current-members

ROS: Robot Operating System



https://www.grandviewresearch.com/industry-analysis/robot-operating-system-market-report

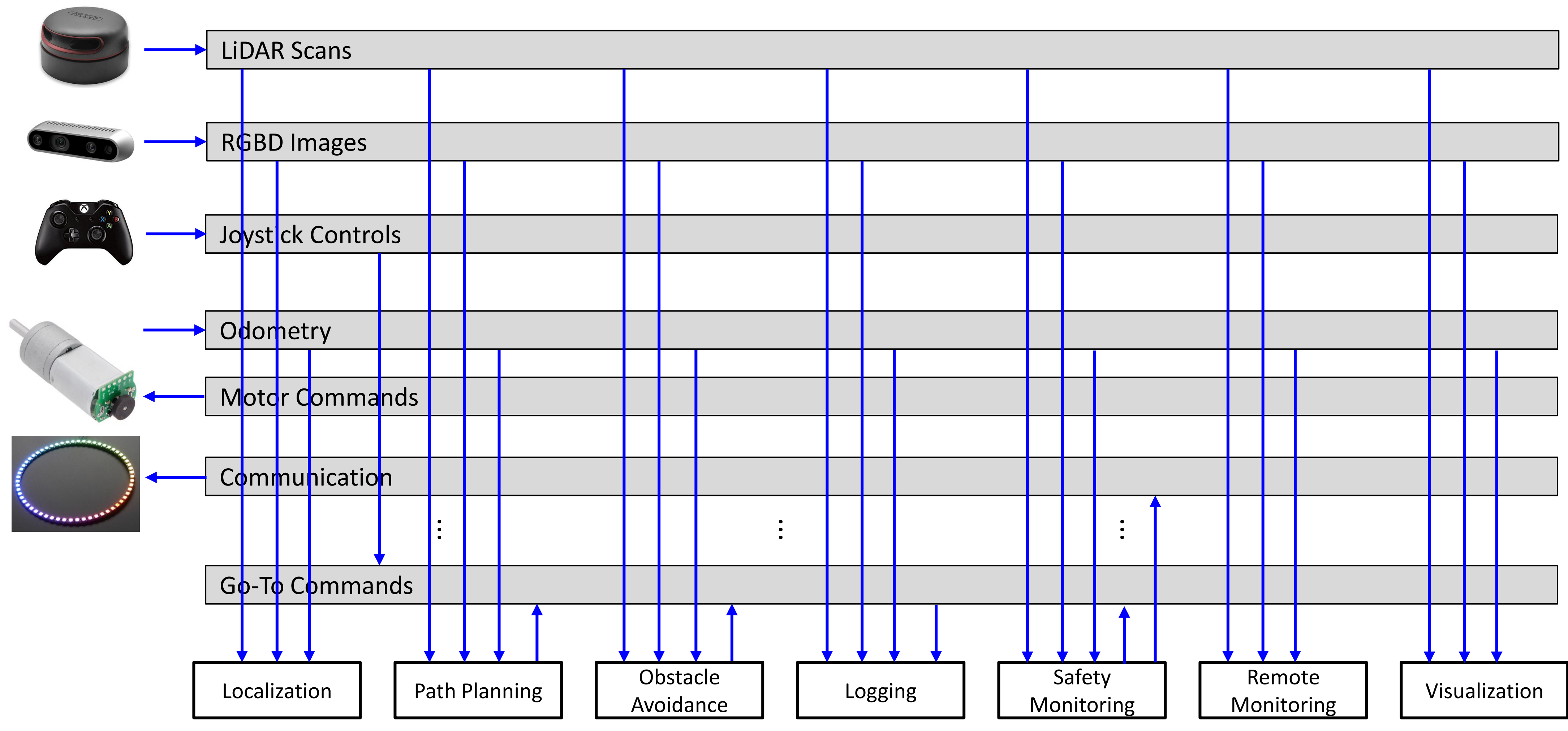
ROS: Robot Operating System

- 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.
- From drivers to state-of-the-art AI algorithms, and with powerful developer tools, ROS has what you need for your next robotics project.
- And it's all open source.

ROS: Robot Operating System

- **Plumbing:** ROS provides messaging infrastructure designed to support the quick and easy construction of distributed computing systems.

ROS: Robot Operating System

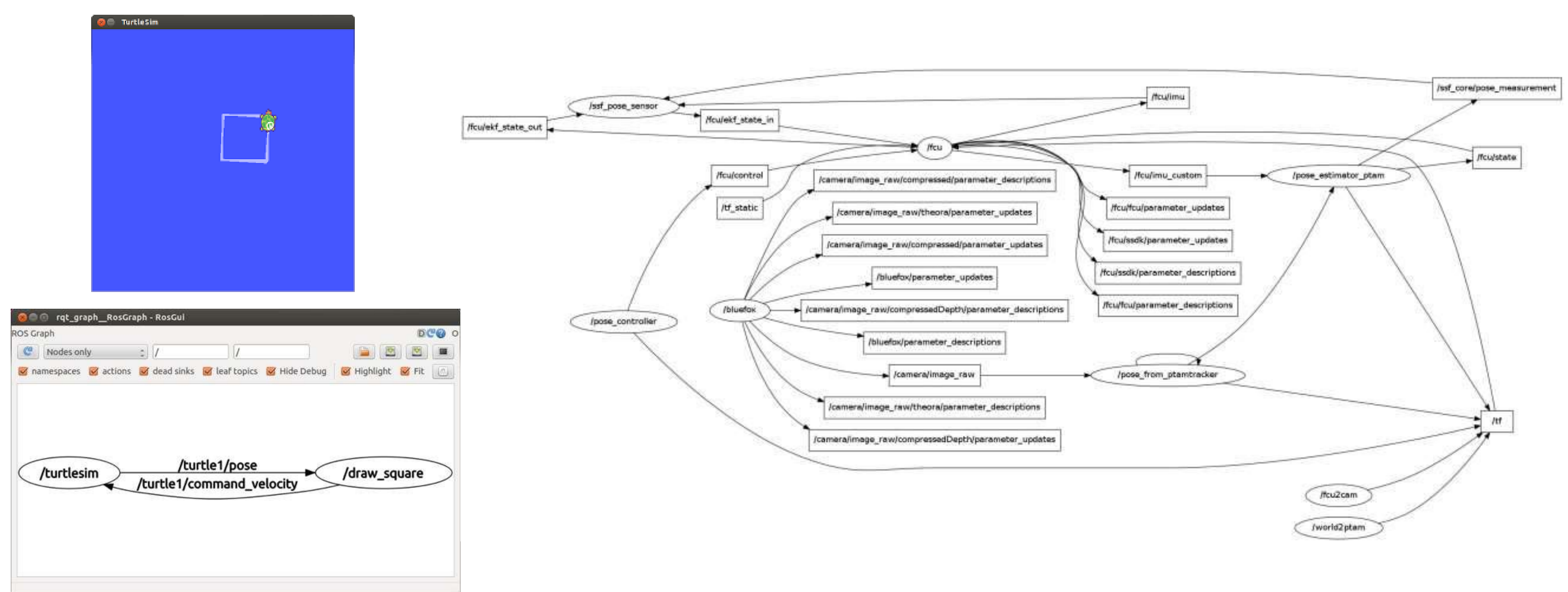


ROS: Robot Operating System

- **Tools:** ROS provides an extensive set of tools for configuring, starting, introspecting, debugging, visualizing, logging, testing, and stopping distributed computing systems.

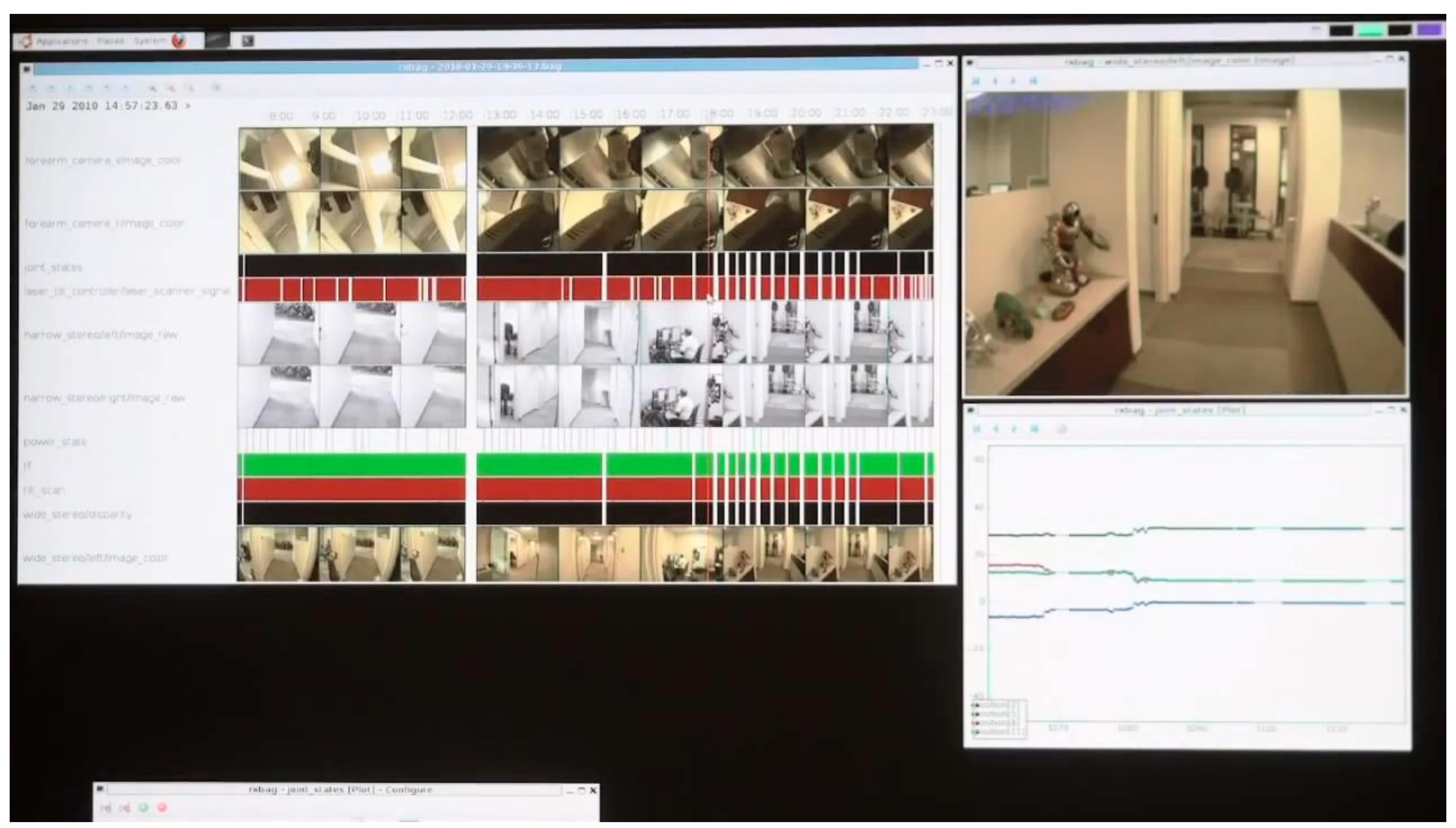
ROS: Robot Operating System

- System visualization: rqt_graph



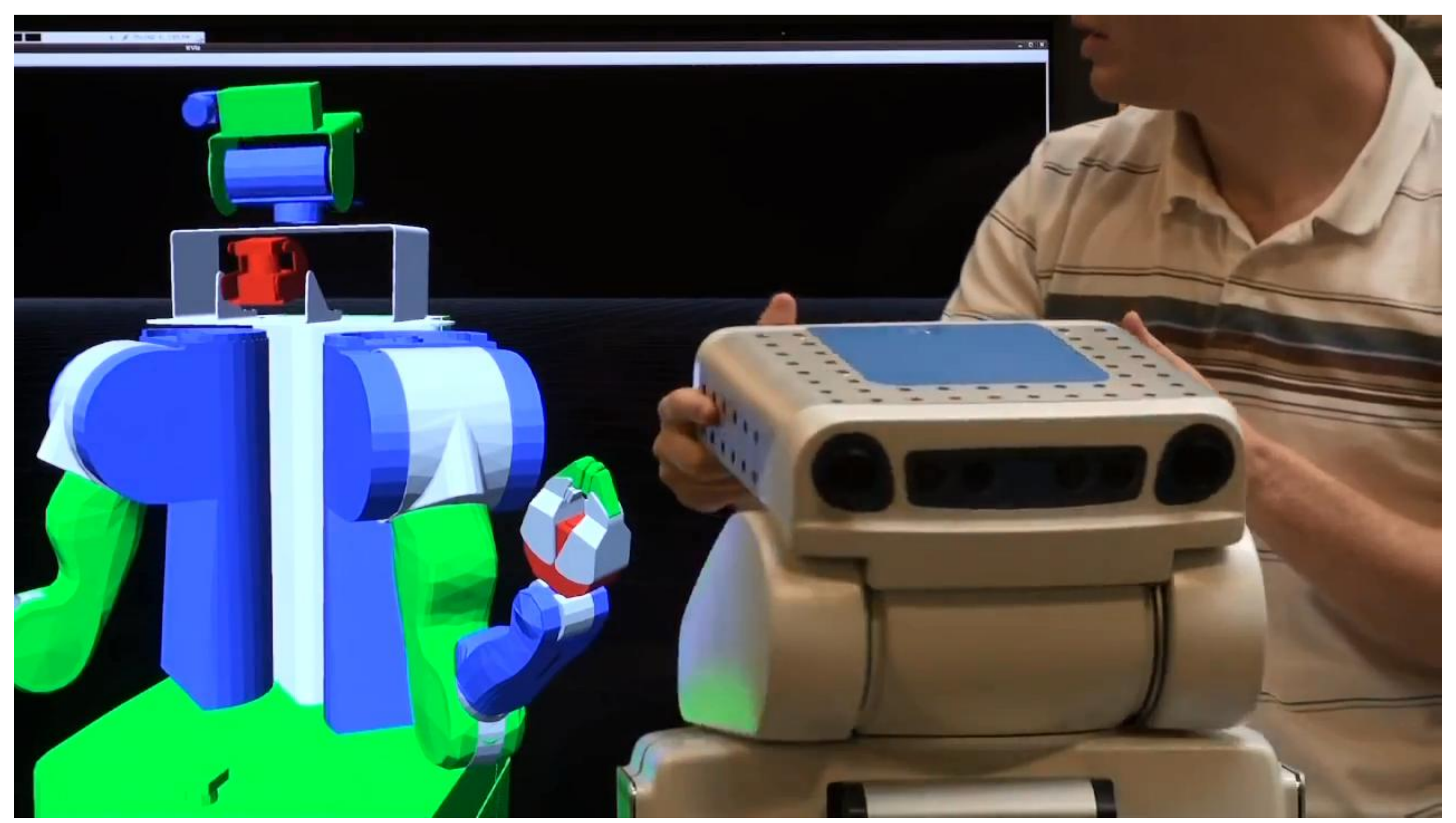
ROS: Robot Operating System

- Logging and visualization of sensor data: rosbag and rqt_bag



ROS: Robot Operating System

- 3D Visualization: RViz



ROS: Robot Operating System

- 3D High-Fidelity Simulation:

DARPA Robotics Challenge: https://www.youtube.com/watch?v=k2wVj0BbtVk

ROS: Robot Operating System

- **Capabilities:** ROS provides a broad collection of libraries that implement useful robot functionality, with a focus on mobility, manipulation, and perception.

CELEBRATING

TEN

YEARS

ROS: Robot Operating System

- **Ecosystem:** ROS is supported and improved by a large community, with a strong focus on integration and documentation. ros.org is a one-stop-shop for finding and learning about the thousands of ROS packages that are available from developers around the world.

ROS: Robot Operating System

- A history of ROS:
  - Early days at Stanford: before 2007
  - Willow Garage: 2007-2013
  - OSRF and Open Robotics: 2013-present
    - Open Source Robotics Foundation (OSRF) changed its name to Open Robotics in 2017
- We will use ROS1 Noetic (+ ROS2 options)

ROS Noetic Ninjemys (Recommended)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013			May, 2015
ROS Groovy Galapagos	December 31, 2012			July, 2014
ROS Fuerte Turtle	April 23, 2012			--
ROS Electric Emys	August 30, 2011			--
ROS Diamondback	March 2, 2011			--
ROS C Turtle	August 2, 2010			--
ROS Box Turtle	March 2, 2010			--

ROS2

Distro	Release date	Logo	EOL date
Kilted Kaiju	May 23, 2025		December 2026
Jazzy Jalisco	May 23, 2024		May 2029
Iron Irwini	May 23, 2023		December 4, 2024
Humble Hawksbill	May 23, 2022		May 2027
Galactic Geochelone	May 23, 2021		December 9, 2022

COMPSCI-603: Robotics

ROS Basics (ROS 1 version)

ROS: Main Concepts

- Nodes
- Topics
- Messages
- Services
- ROS Master
- Parameters
- Packages

ROS: Main Concepts

- **Nodes**
  - Single-purposed executable programs that perform computation.
    - E.g., sensor driver(s), actuator driver(s), mapper, planner, UI, etc.
  - Each node has a unique name.
  - Individually compiled, executed, and managed.
  - Nodes are written using a ROS client library.
    - rospy – python client library (used in this course)
    - roscpp – C++ client library
  - Nodes can publish or subscribe to Topics.
  - Nodes can also provide or use Services.

ROS: Main Concepts

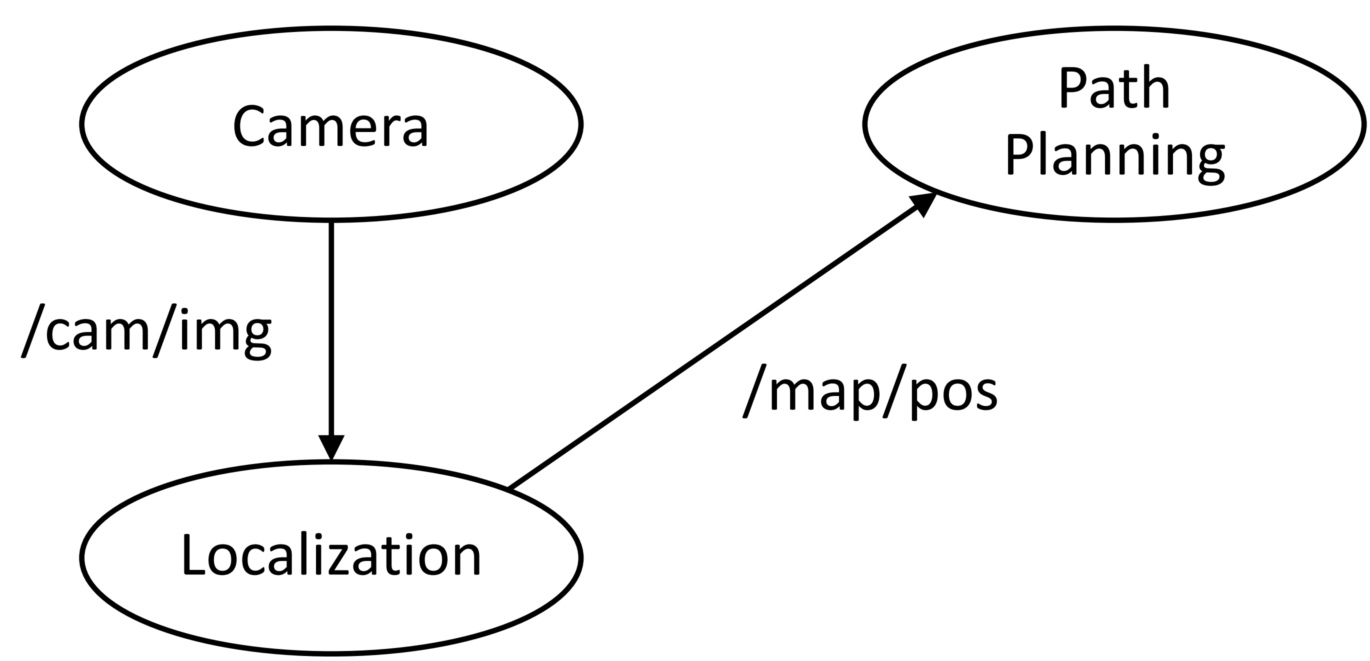
- **Nodes**
  - Single-purposed executable programs that perform computation.

ROS: Main Concepts

- **Topics**
  - Names for a stream of broadcasting messages.
  - Each topic is uniquely identifiable by its name.
  - Each topic has a defined message type.
    - E.g., data from a laser range-finder might be sent on a topic called `scan`, with a message type of `LaserScan`.
  - Topics implement a Publish/Subscribe model: 1-to-N broadcasting.
  - Nodes communicate with each other by publishing and receiving messages from topics.

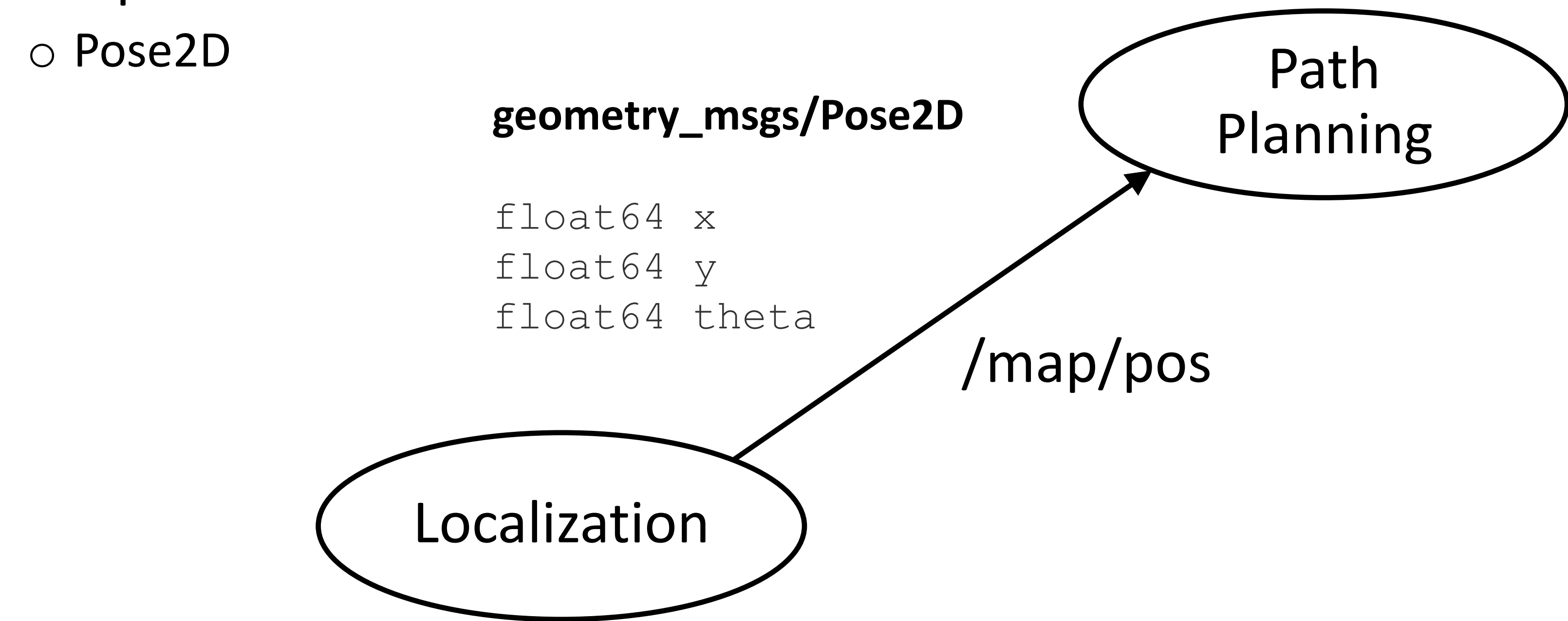
ROS: Main Concepts

- **Topics**
  - Names for a stream of broadcasting messages.



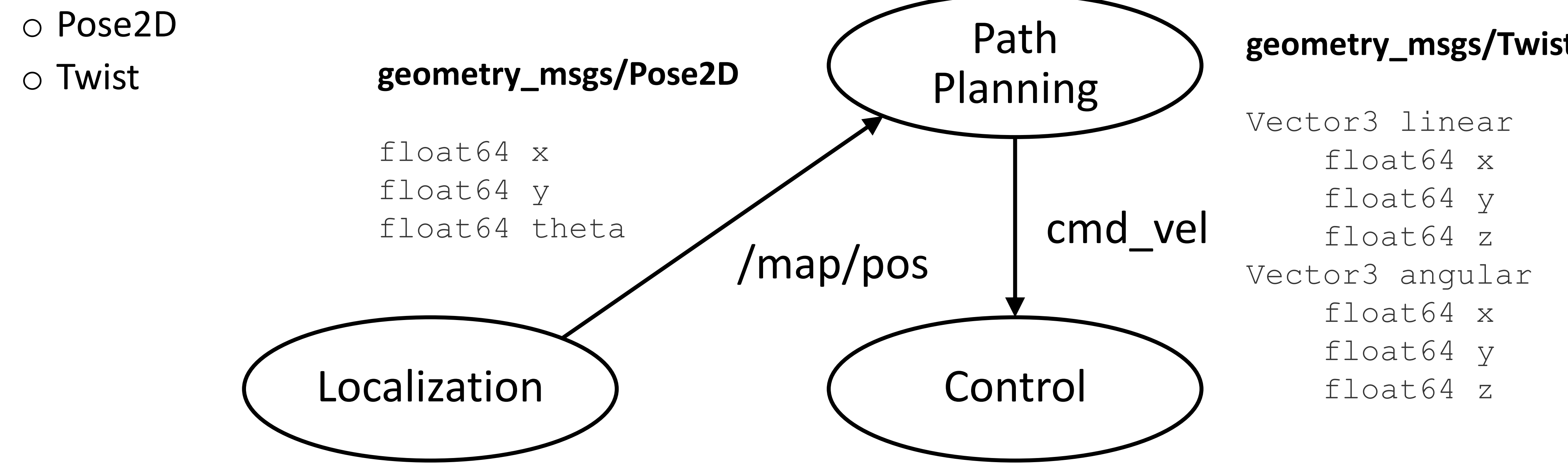
ROS: Main Concepts

- **Messages**
  - Strictly-typed data structures for inter-node communication.
  - Language agnostic data structures, so that Python nodes can talk to C++ nodes.
  - Examples:
    - Pose2D



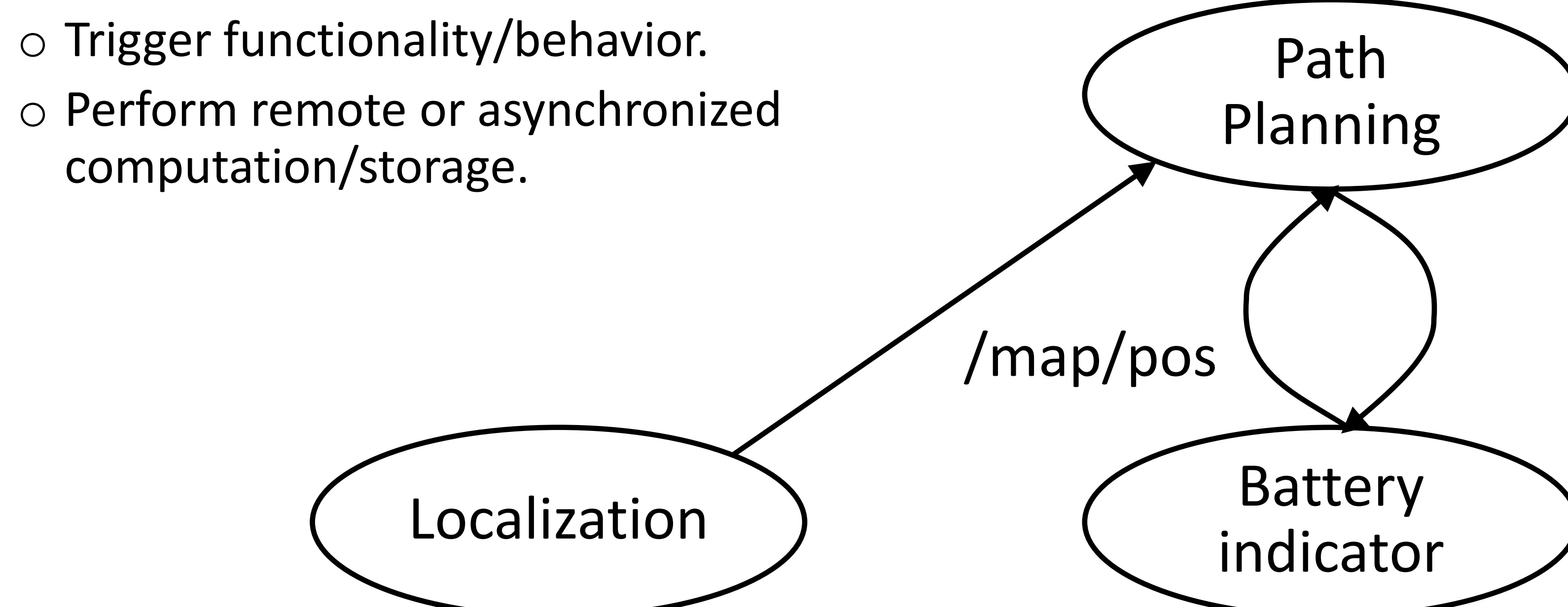
ROS: Main Concepts

- **Messages**
  - Strictly-typed data structures for inter-node communication.
  - Language agnostic data structures, so that Python nodes can talk to C++ nodes.
  - Examples:
    - Pose2D
    - Twist



ROS: Main Concepts

- **Services**
  - Inter-node transactions by request and response.
  - Service/Client model to enable 1-to-1 request-response.
  - Service roles:
    - Trigger functionality/behavior.
    - Perform remote or asynchronized computation/storage.

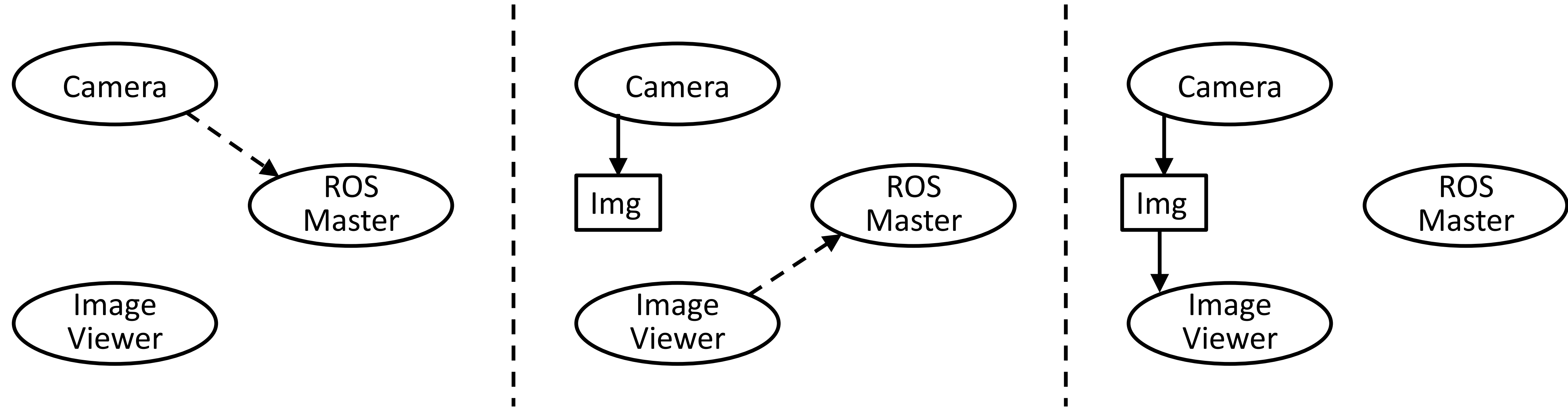


ROS: Main Concepts

- **ROS Master** (ROS1 only)
    - Provides name registration and lookup to nodes.
    - Every node connects to the ROS Master at startup to register details of the message streams they publish, and the streams to which that they to subscribe.
    - When a new node appears, ROS Master provides it with the information that it needs to form a direct peer-to-peer connection with other nodes publishing and subscribing to the same message topics.
    - Without the ROS Master, nodes would not be able to find each other, exchange messages, or invoke services.
        - For this reason, ROS (more accurately ROS1) is a *centralized* system.

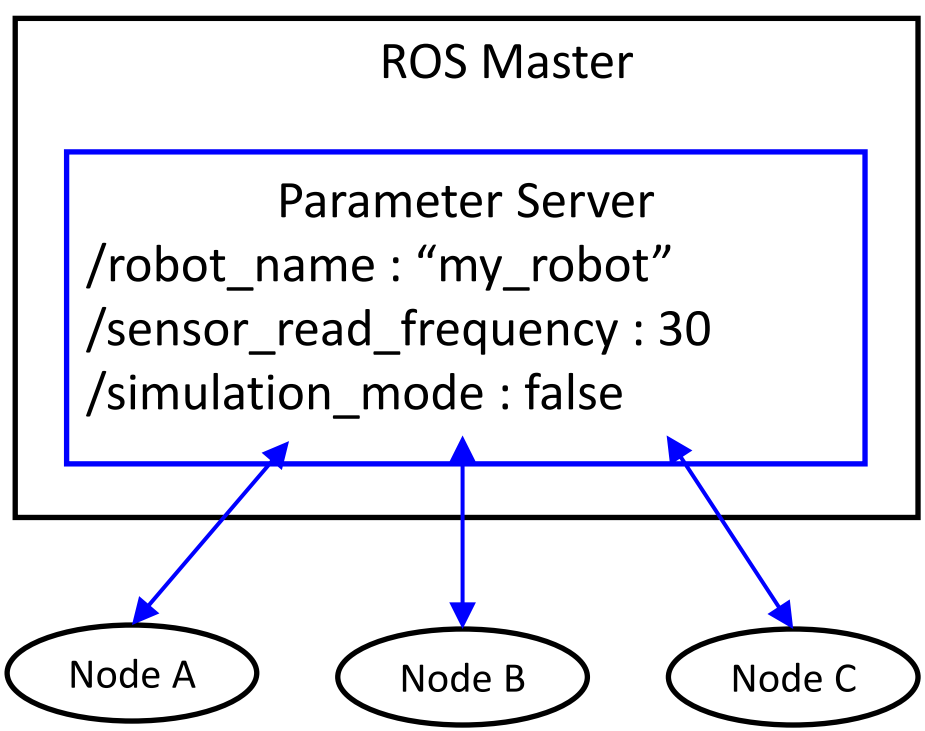
ROS: Main Concepts

- **ROS Master**
  - Provides name registration and lookup to nodes.



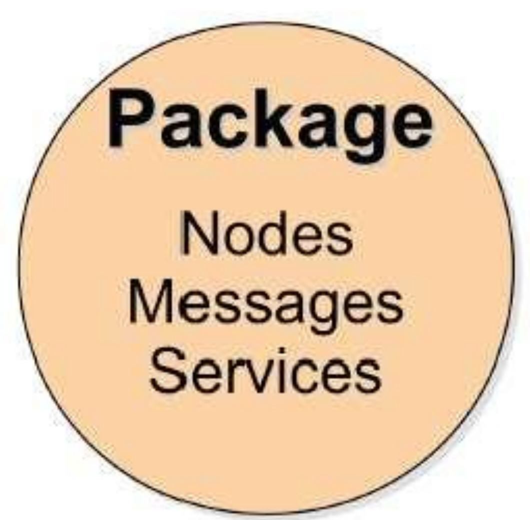
ROS: Main Concepts

- **Parameter Server**
  - Is typically used to save configuration information in ROS.
  - A shared, multi-variate dictionary that is accessible via network APIs.
  - Nodes use this server to store and retrieve parameters at runtime.
  - Best used for static data such as configuration parameters.
  - Runs inside the ROS master.



ROS: Main Concepts

- **Packages**
  - Software programs in ROS are organized in packages.
  - A package contains one or more nodes, and provides a ROS interface with package information



ROS: Main Concepts

- **ROS Environment**
  - ROS relies on the notion of combining spaces using the shell environment.
    - This makes developing against different versions of ROS or against different sets of packages easier.
  - After you install ROS you will have to setup.*sh files in '/opt/ros/<distro>/', and you could source them like so:

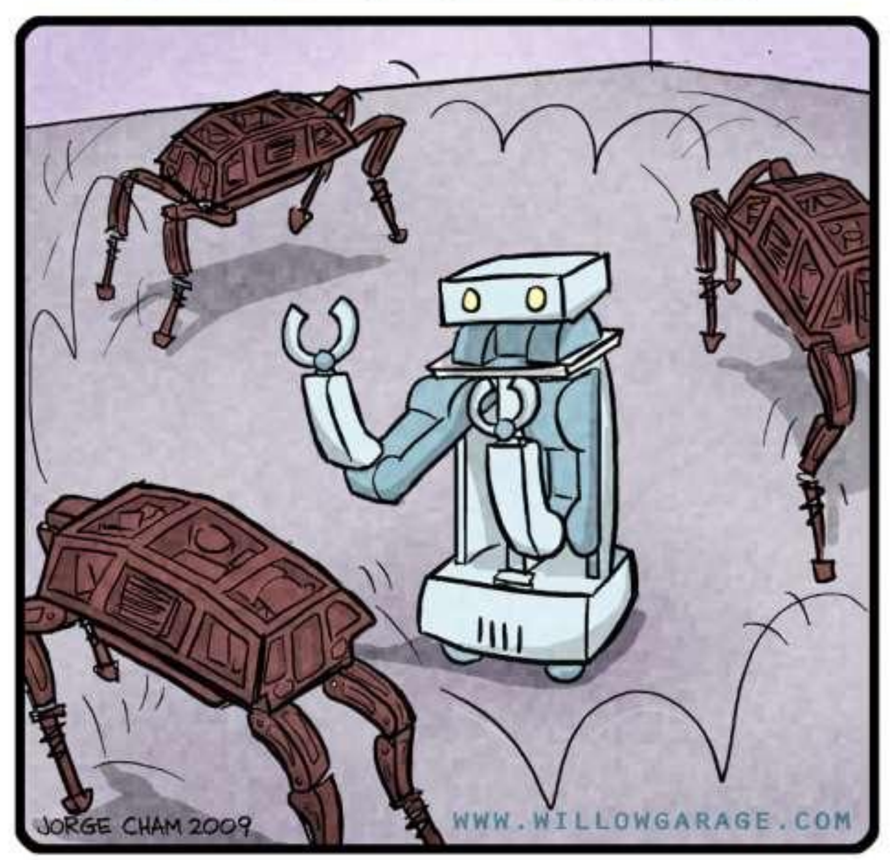
```
$ source /opt/ros/noetic/setup.bash
```

  - You will need to run this command on every new shell you open to have access to the ROS commands, unless you add this line to your bash startup file (~/.bashrc).

Example Commands

- roscore (for ROS1)
- rosrun
- rosnode
- rostopic

http://wiki.ros.org/ROS/CommandLineTools



Example Commands

- roscore
  - `roscore` is the first thing you should run when using ROS1:

    `$ roscore`

  - `roscore` will start up:
    - a ROS Master
    - a ROS Parameter Server
    - a rosout logging node

```
compsci603@compsci603ubuntu2004:~/catkin_ws$ roscore
... logging to /home/compsci603/.ros/log/e165025c-afd9-11ed-8d5a-07c22f6a93ff/ros
launch-compsci603ubuntu2004-7870.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://compsci603ubuntu2004:40973/
ros_comm version 1.15.15


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.15

NODES

auto-starting new master
process[master]: started with pid [7878]
ROS_MASTER_URI=http://compsci603ubuntu2004:11311/

setting /run_id to e165025c-afd9-11ed-8d5a-07c22f6a93ff
process[rosout-1]: started with pid [7888]
started core service [/rosout]
```

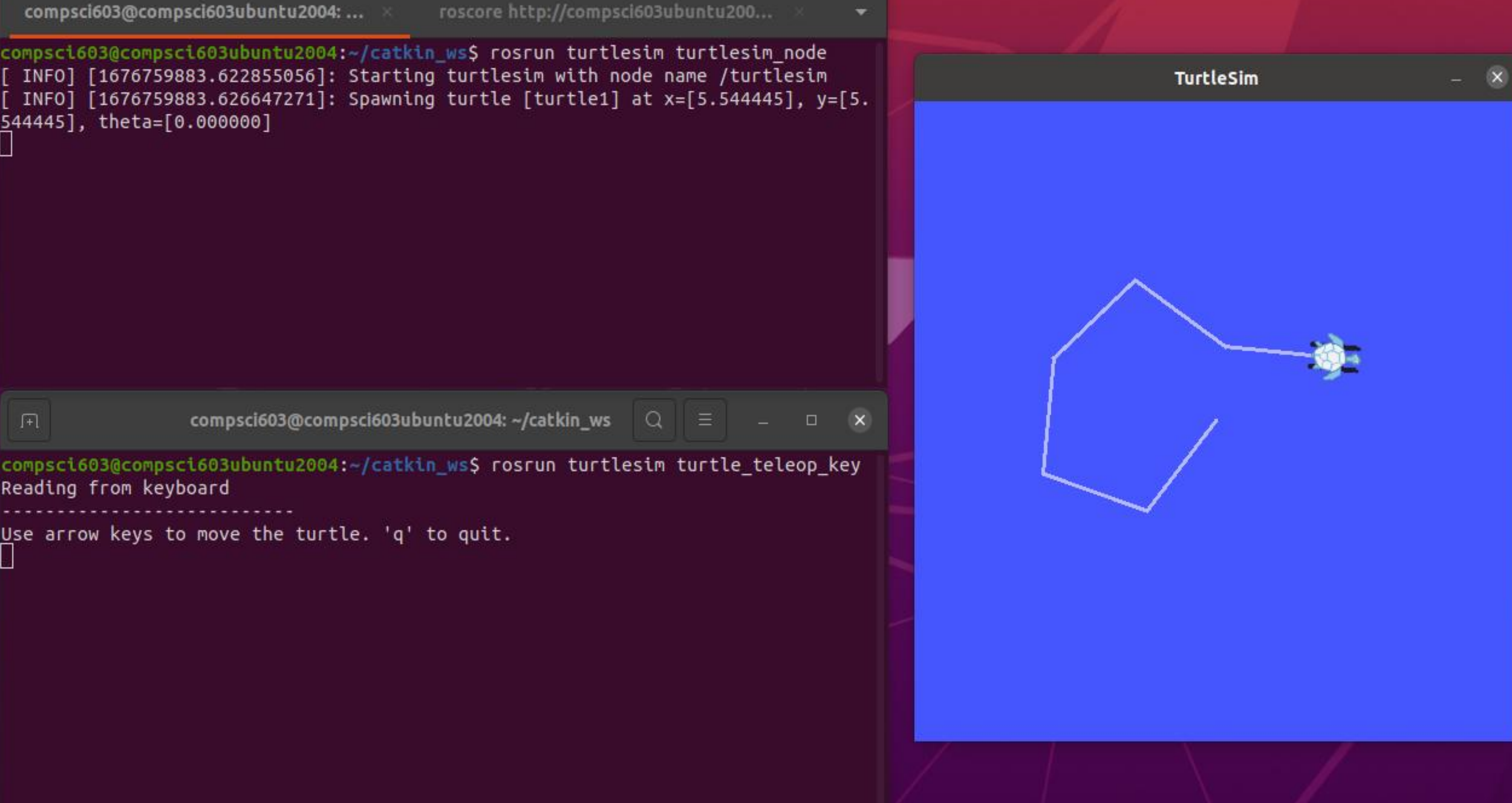
Example Commands

- rosrun
  - Execute a node in a package using:

```
$ rosrun <package> <executable>
```

  - Example: In separate terminals, run:

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


compsci603@compsci603ubuntu2004: ...
roscore http://compsci603ubuntu200...
compsci603@compsci603ubuntu2004:~/catkin_ws$ rosrun turtlesim turtlesim_node
[INFO] [1676759883.622855056]: Starting turtlesim with node name /turtlesim
[INFO] [1676759883.626647271]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
compsci603@compsci603ubuntu2004: ~/catkin_ws
compsci603@compsci603ubuntu2004:~/catkin_ws$ rosrun turtlesim turtle_teleop_key
Reading from keyboard

Use arrow keys to move the turtle. 'q' to quit.
TurtleSim

Example Commands

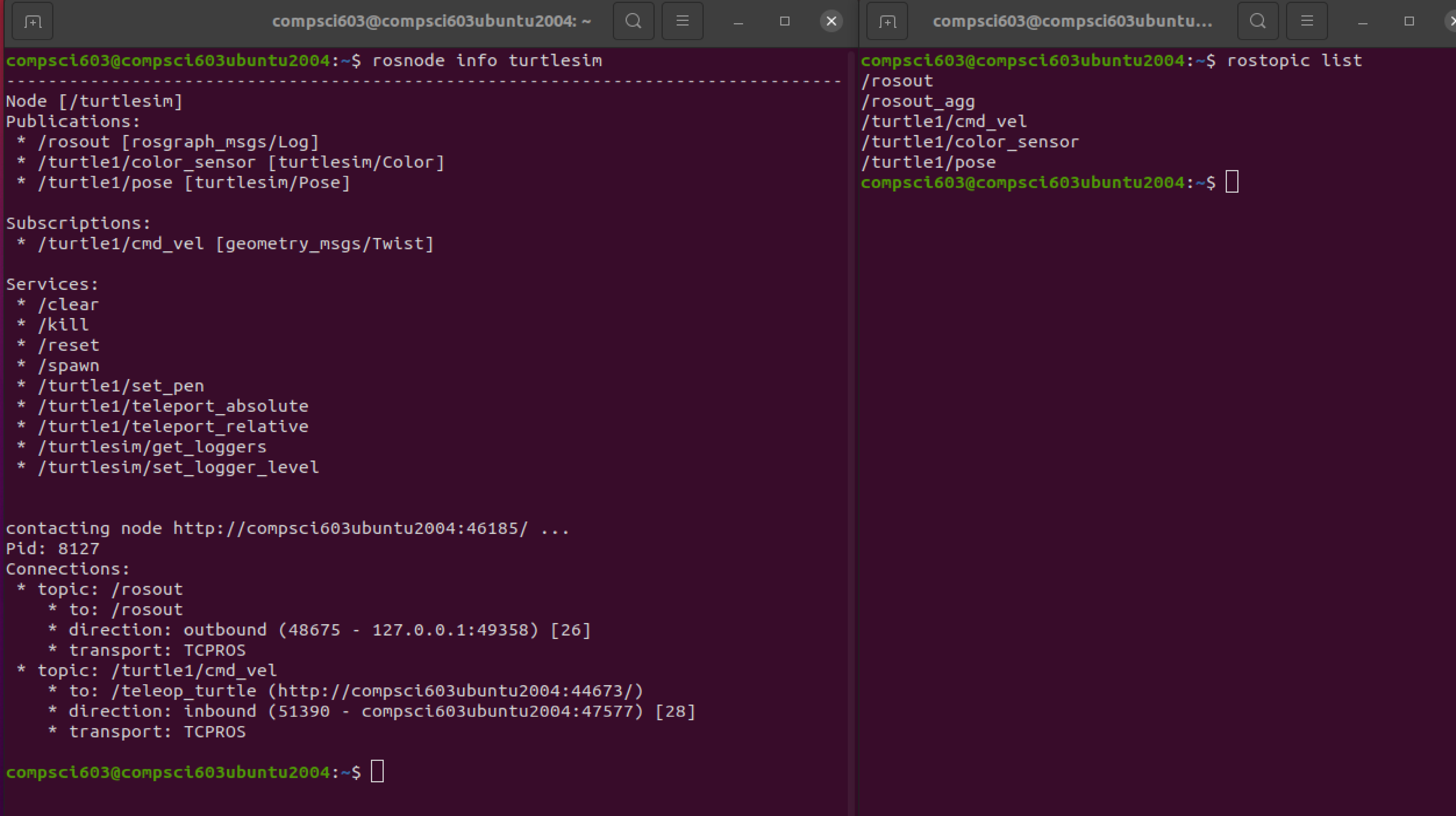
- rosnode: Provides debugging information about ROS nodes, including publication, subscription and connection.

Command	
$rosnode list	List active nodes
$rosnode ping	Test connectivity to node
$rosnode info	Print information about a node
$rosnode kill	Kill a running node
$rosnode machine	List nodes running on a particular machine

Example Commands

- rostopic: Gives information about a topic and allows to publish messages on a topic.

Command	
$rostopic list	List active topics
$rostopic echo /topic	Prints messages of the topic to the screen
$rostopic info /topic	Print information about a topic
$rostopic type /topic	Prints the type of messages the topic publishes
$rostopic pub /topic type args	Publishes data to a topic


compsci603@compsci603ubuntu2004: ~
compsci603@compsci603ubuntu2004:~$ rosnode info turtlesim

Node [/turtlesim]
Publications:
* /rosout [rosgraph_msgs/Log]
* /turtle1/color_sensor [turtlesim/Color]
* /turtle1/pose [turtlesim/Pose]
Subscriptions:
* /turtle1/cmd_vel [geometry_msgs/Twist]
Services:
* /clear
* /kill
* /reset
* /spawn
* /turtle1/set_pen
* /turtle1/teleport_absolute
* /turtle1/teleport_relative
* /turtlesim/get_loggers
* /turtlesim/set_logger_level
contacting node http://compsci603ubuntu2004:46185/ ...
Pid: 8127
Connections:
* topic: /rosout
* to: /rosout
* direction: outbound (48675 - 127.0.0.1:49358) [26]
* transport: TCPROS
* topic: /turtle1/cmd_vel
* to: /teleop_turtle (http://compsci603ubuntu2004:44673/)
* direction: inbound (51390 - compsci603ubuntu2004:47577) [28]
* transport: TCPROS
compsci603@compsci603ubuntu2004:~$
compsci603@compsci603ubuntu...
compsci603@compsci603ubuntu2004:~$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
compsci603@compsci603ubuntu2004:~$

ROS: Common Commands

- rostopic pub: Allows to publish messages to a topic.
- For example, to make the turtle move forward at a 0.2m/s speed, you can publish a cmd_vel message to the topic /turtle1/cmd_vel:

Topic Type

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist
'{linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0}}'
```

Arguments

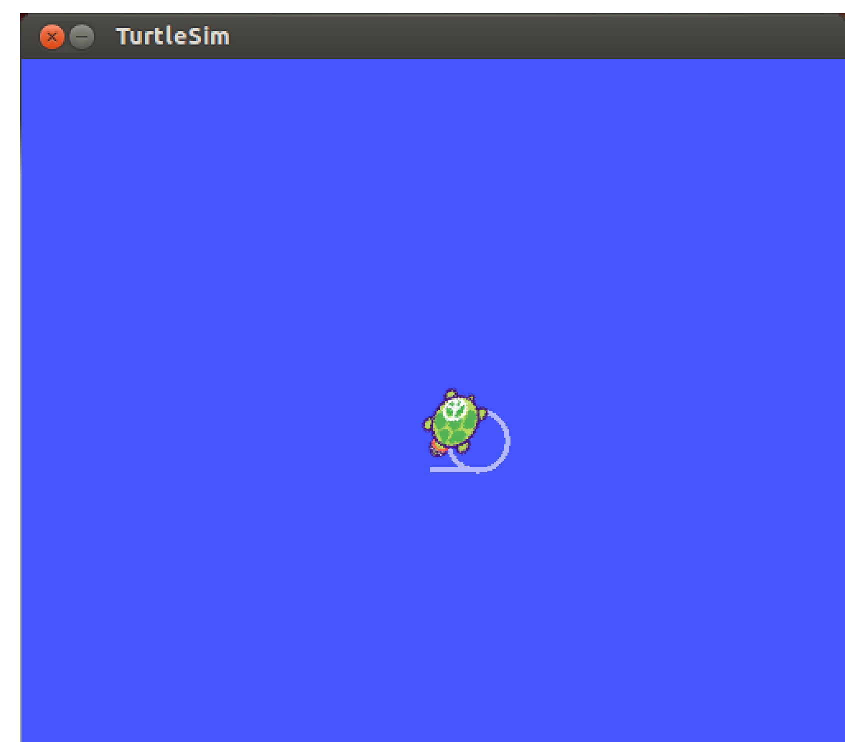
- To specify only the linear velocity along the x-axis:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist
'{linear: {x: 0.2}}'
```

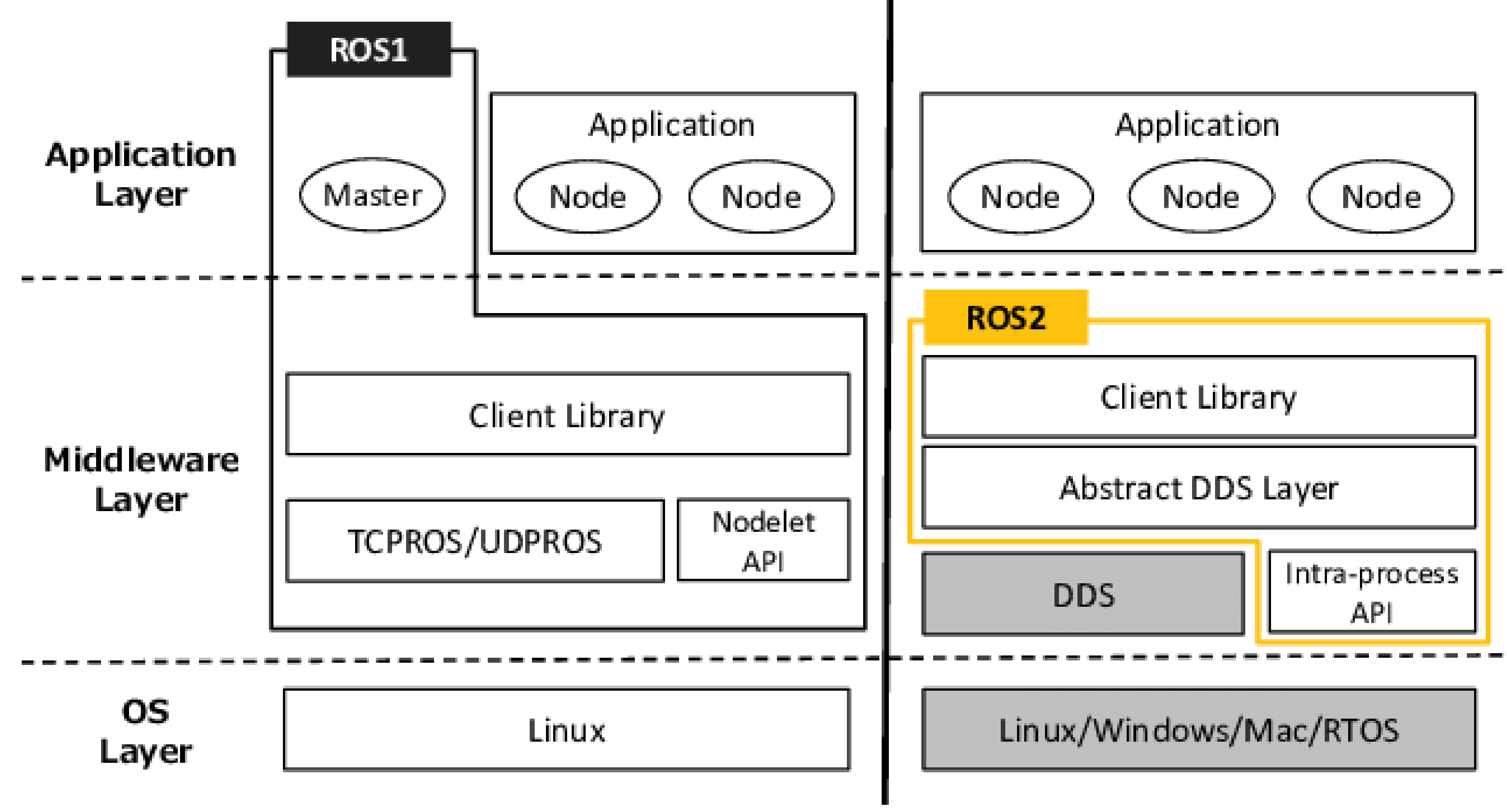
ROS: Common Commands

- rostopic pub
  - Some of the messages like `cmd_vel` have a predefined timeout.
  - To publish a message continuously, use the argument `-r` with the loop rate in Hz.
  - For example, to make the turtle turn in circles continuously, type:

```
$ rostopic pub /turtle1/cmd_vel -r 10
geometry_msgs/Twist  '{angular: {z: 0.5}}'
```

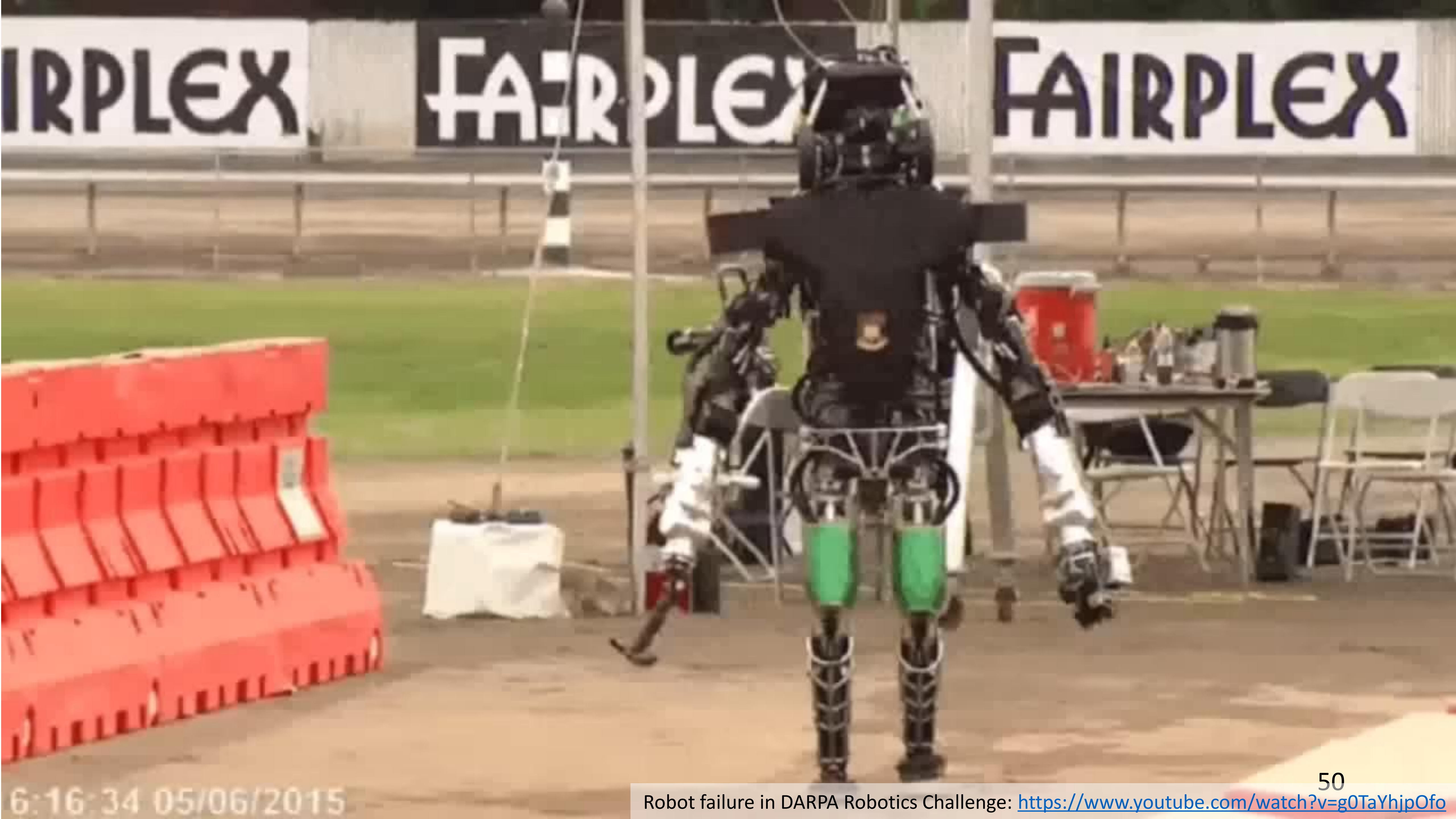


ROS 1 vs ROS 2



DDS (Data Distribution Service): a networking middleware that simplifies complex network programming

Image credit: Maruyama, Y., Kato, S. and Azumi, T., 2016, October. Exploring the performance of ROS2. In International Conference on Embedded Software (pp. 1-10).



Robot failure in DARPA Robotics Challenge: https://www.youtube.com/watch?v=g0TaYhjpOfo

COMPSCI-603: Robotics

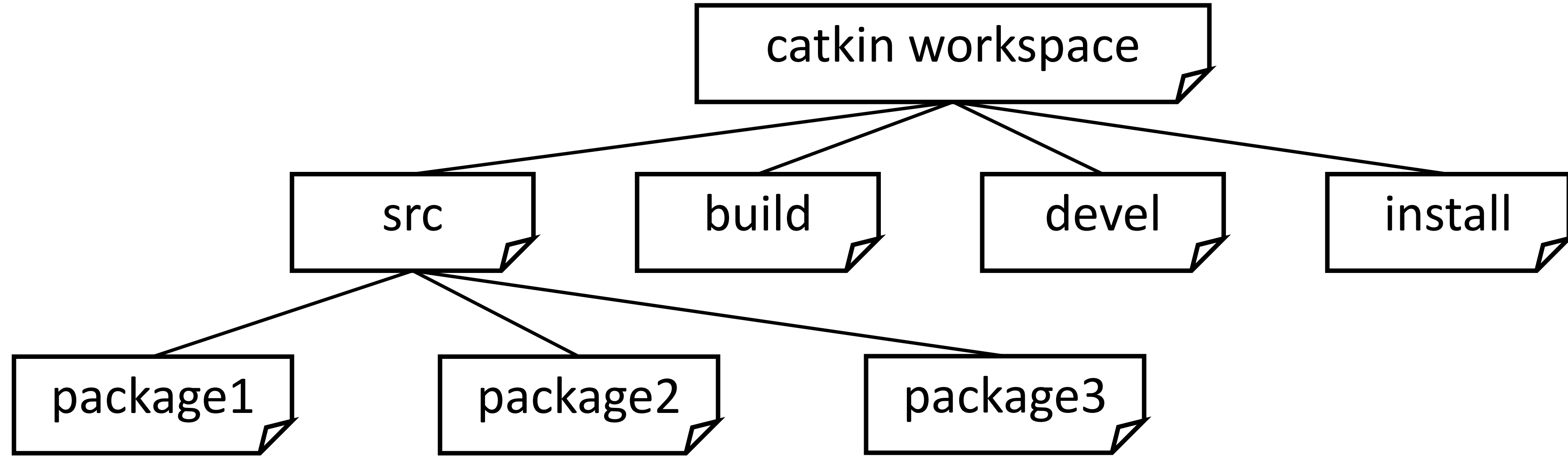
Overview of ROS Programming in Python
(ROS 1 version)

catkin Build System

- catkin is the ROS build system.
  - The set of tools that ROS uses to generate executable programs, libraries and interfaces.

- Implemented as custom CMake macros along with some Python code.
  - CMake is an open-source, cross-platform family of tools designed to build, test and package software.

catkin Workspace



Source space	It contains the source code of catkin packages. Each folder within the source space contains one or more packages.
Build Space	It 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.
Development Space	It is where built targets are placed prior to being installed.
Install Space	Once targets are built, they can be installed into the install space by invoking the install target.

catkin Workspace

- To create and build a catkin workspace, run:

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

- To install final packages:

```
$ cd ~/catkin_ws/build
$ make install
```

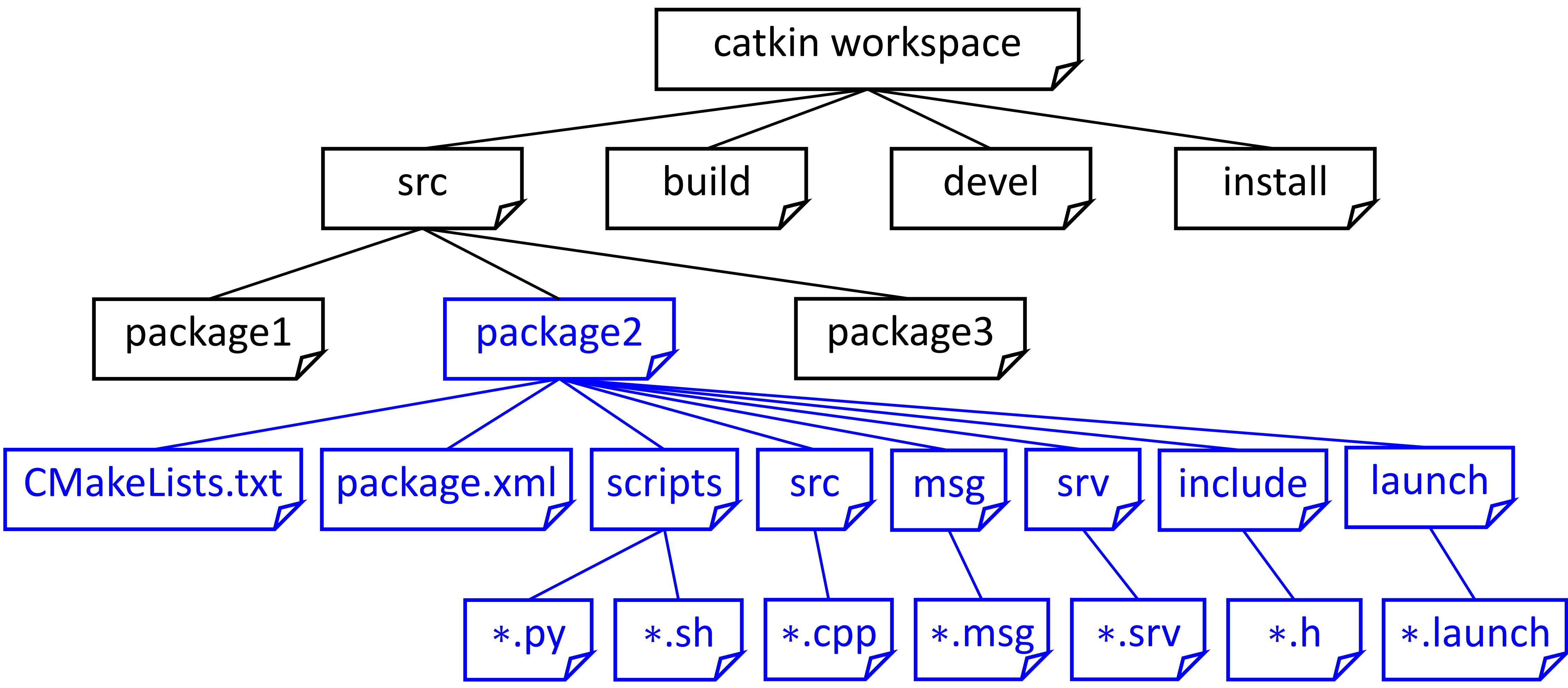
More info:
http://wiki.ros.org/catkin/Tutorials/using_a_workspace

```
catkin_ws/                -- WORKSPACE
  src/                    -- SOURCE SPACE
    ...
  build/                  -- BUILD SPACE
  devel/                  -- DEVEL SPACE
    setup.bash           \
    setup.sh             |-- Environment setup files
    setup.zsh            /
    etc/                  -- Generated configuration files
    include/              -- Generated header files
    lib/                  -- Generated libraries and other artifacts
      package_1/
        bin/
        etc/
        include/
        lib/
        share/
        ...
      package_n/
        bin/
        etc/
        include/
        lib/
        share/
    share/                -- Generated architecture independent artifacts
    ...
```

ROS Packages

- ROS software is organized into packages, each of which contains a combination of code, data, and documentation.
- A ROS package is placed as a directory inside a catkin workspace that has a package.xml file in it.
- A package contains the source files for one node or more, and configuration files.
- Packages are the **most atomic unit of build and the unit of release.**

ROS Packages



ROS Packages

- package.xml defines properties of the package:
  - the package name, version numbers, authors, dependencies on other catkin packages, and more.

```
1  <?xml version="1.0"?>
2  <package format="2">
3    <name>beginner_tutorials</name>
4    <version>0.1.0</version>
5    <description>The beginner_tutorials package</description>
6
7    <maintainer email="you@yourdomain.tld">Your Name</maintainer>
8    <license>BSD</license>
9    <url type="website">http://wiki.ros.org/beginner_tutorials</url>
10   <author email="you@yourdomain.tld">Jane Doe</author>
11
12   <buildtool_depend>catkin</buildtool_depend>
13
14   <build_depend>roscpp</build_depend>
15   <build_depend>rospy</build_depend>
16   <build_depend>std_msgs</build_depend>
17
18   <exec_depend>roscpp</exec_depend>
19   <exec_depend>rospy</exec_depend>
20   <exec_depend>std_msgs</exec_depend>
21
22 </package>
```

ROS Packages

- To create a ROS package, change to the source directory of the workspace: `$ cd ~/catkin_ws/src`
- catkin_create_pkg creates a new package with the specified dependencies:

```
$ catkin_create_pkg <package_name> [depend1] depend2] [depend3]
```

- For example, to create a package named first_pkg, run:

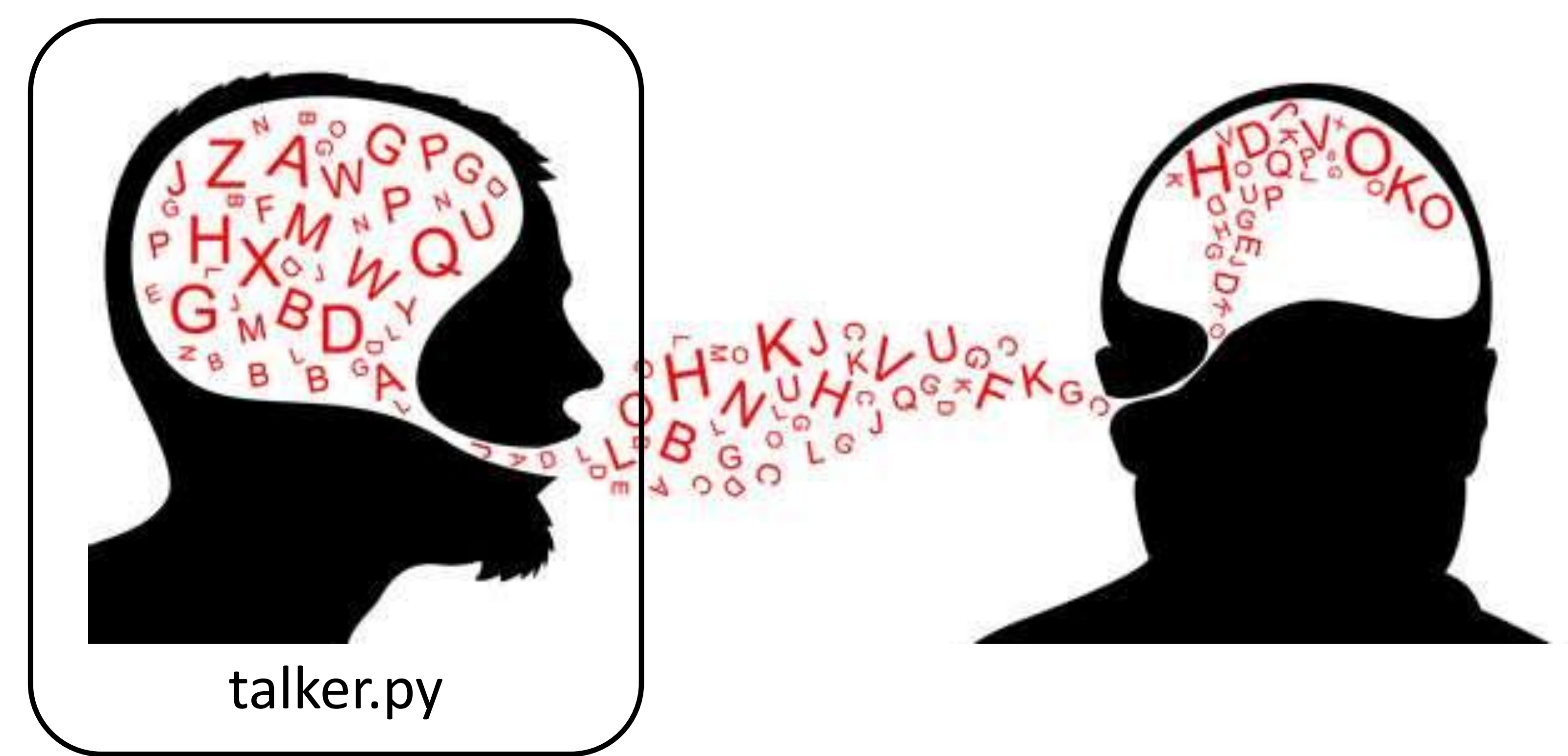
```
$ catkin_create_pkg first_pkg std_msgs rospy roscpp
```

- Then, catkin_make the workspace and **don’t forget to add the env:**

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

ROS Example: Talker and Listener

- We will create a package with two nodes:
  - *talker* publishes messages to topic *chatter*.
  - *listener* reads the messages from the topic and prints them out to the screen.



ROS Example: Talker Node

- ROS Noetic uses Python 3.8
- Earlier versions used Python 2:
  `#!/usr/bin/env python`

- Within a package folder, create Python scripts in the “scripts” folder:

```
$ roscd first_pkg
$ mkdir scripts
$ cd scripts
$ vi talker.py
```

- After editing, make it executable:

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

```python
#!/usr/bin/env python3

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
```

ROS Example: Talker Node

- Import `rospy` when writing a Python ROS Node.
  - `rospy` is a pure Python client library for ROS.
  - `rospy` favors implementation speed (i.e., developer time) over runtime performance.

- Exception block monitors interruption exceptions:
  - Ctrl-C
  - Node shutdown
  - Node sleep

```
#!/usr/bin/env python3

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
```

ROS Example: Talker Node

- This line of code creates a ROS node.
- The argument `talker` is the name of the node.
- `anonymous=True` adds random numbers to the end of the node name to ensure it is unique.
  - Spawn multiple copies of the node without collision.

```
#!/usr/bin/env python3

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
```

ROS Example: Talker Node

- This declares that the node publishes to the `chatter` topic using the message type `String`.
- It registers the topic in ROS Master and manage the advertisement.
- It has three parameters:
    1. Topic name
    2. ROS message type (NOT Python type)
    3. Queue size (buffer size)

```
#!/usr/bin/env python3

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
```

ROS Example: Talker Node

- This publishes a message with the type `String` to the topic named `chatter`.
- The message's type must strictly agree with the type given as a type parameter to the Publisher call.

```
#!/usr/bin/env python3

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
```

ROS Example: Talker Node

- `rospy.Rate()` is designed to run loops at a desired frequency (in Hz).
- `rospy.sleep()` method
  - Sleeps for any leftover time in a cycle.
  - Is calculated from the last time sleep, reset, or when the constructor was called.

```
#!/usr/bin/env python3

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
```

ROS Example: Talker Node

- `rospy.is_shutdown()` is used to check if the node is shutdown (to determine whether the node should continue running).
- It returns true if:
  - Ctrl-C is received.
  - the node is kicked off the network by another node with the same name.
  - `rospy.shutdown()` is called by another part of the program.
  - The node is destroyed.

```
#!/usr/bin/env python3

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
```

ROS Example: Talker Node

- `rospy.loginfo(str)` is ROS logging that
  - Prints the messages to the screen.
  - Writes them to the node's log file.
  - Writes them to `rosout`.

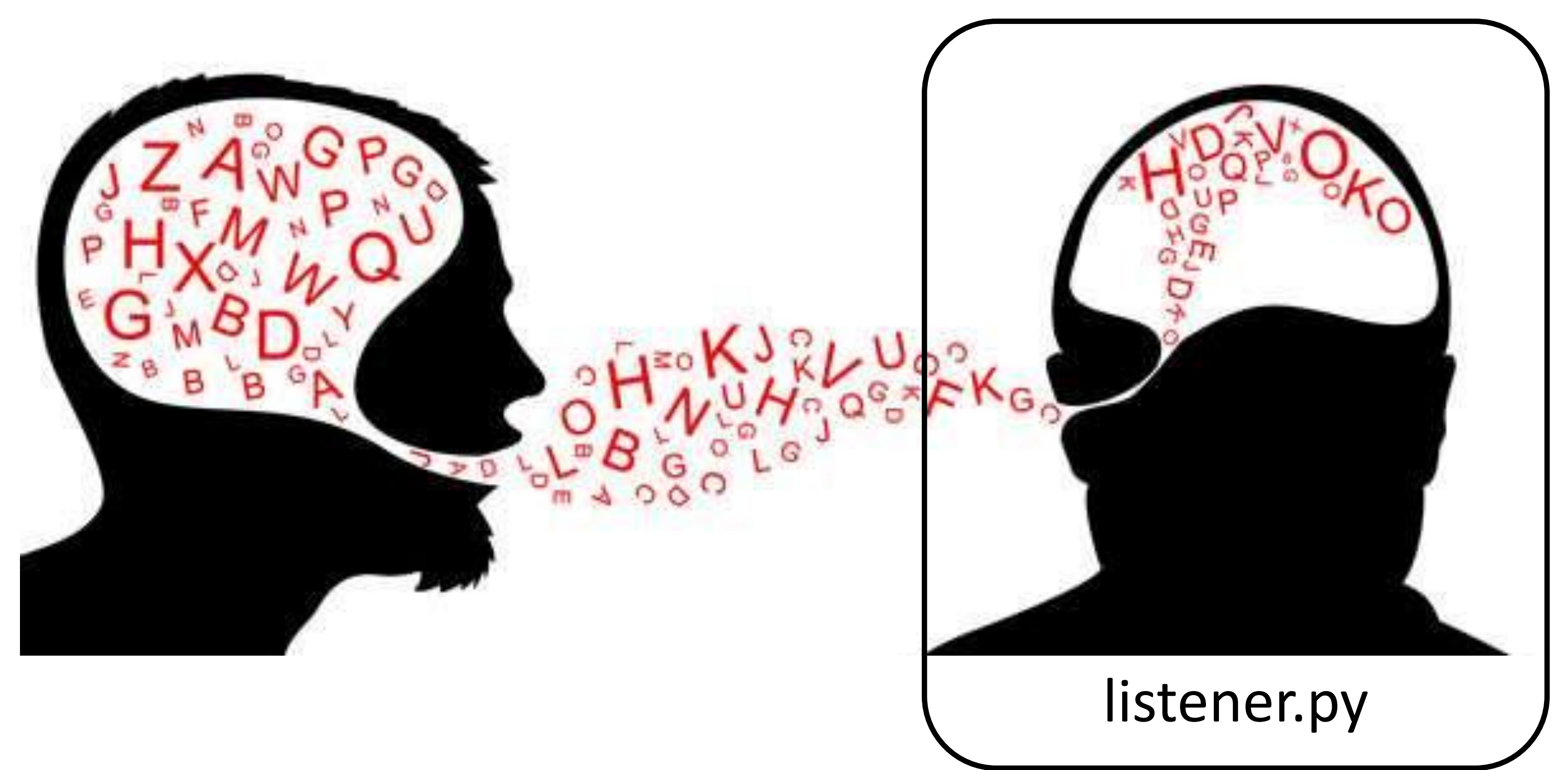
	Debug	Info	Warn	Error	Fatal
stdout		X			
stderr			X	X	X
log file	X	X	X	X	X
/rosout	X	X	X	X	X

```
#!/usr/bin/env python3

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
```

```
rospy.logdebug(hello_str)
rospy.logwarn(hello_str)
rospy.logerr(hello_str)
rospy.logfatal(hello_str)
```

ROS Example: Talker and Listener

- We will create a package with two nodes:
  - *talker* publishes messages to topic *chatter*.
  - *listener* reads the messages from the topic and prints them out to the screen.



ROS Example: Listener Node

- `Rospy.Subscriber()` declares a subscriber to listen to a topic `chatter`.
- It has three parameters:
  1. Topic name
  2. ROS message type
  3. Callback function to handle the message

```
#!/usr/bin/env python3

import rospy
from std_msgs.msg import String

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

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

ROS Example: Listener Node

- When a new messages is received, callback is invoked with the message as the argument.
- Note that actual message content is in the .data attribute:

File: std_msgs/String.msg

Raw Message Definition

```
string data
```

```
#!/usr/bin/env python3

import rospy
from std_msgs.msg import String

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

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

ROS Example: Listener Node

- `rospy.spin()` keeps the node from exiting until the node has been shutdown.
- It is usually used when the main function does not do any other processing.

```
#!/usr/bin/env python3

import rospy
from std_msgs.msg import String

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

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

ROS Example: Listener Node

- The exception handling block is not necessary for a pure subscriber since it does not generate data to the system.

```
#!/usr/bin/env python3

import rospy
from std_msgs.msg import String

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

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

ROS Example: Talker and Listener

- To build your nodes, run:

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

- To run both nodes, in three separate terminal windows, run:

Win 1: `$ roscore`
Win 2: `$ rosrun first_pkg talker.py`
Win 3: `$ rosrun first_pkg listener.py`

compsci603@compsci603ubuntu2004: ~/catkin_ws

compsci603@compsci603ubuntu2004: ~/catkin_ws | roscore http://compsci603ubuntu2004:11311/

```
[INFO] [1676758538.565952]: hello world 1676758538.5658426
[INFO] [1676758538.664186]: hello world 1676758538.664074
[INFO] [1676758538.766327]: hello world 1676758538.7662187
[INFO] [1676758538.864476]: hello world 1676758538.8643656
[INFO] [1676758538.963542]: hello world 1676758538.9634342
[INFO] [1676758539.064500]: hello world 1676758539.0643911
[INFO] [1676758539.166392]: hello world 1676758539.1662862
[INFO] [1676758539.267730]: hello world 1676758539.267621
[INFO] [1676758539.364964]: hello world 1676758539.3648562
[INFO] [1676758539.464256]: hello world 1676758539.4641485
[INFO] [1676758539.564257]: hello world 1676758539.5641482
[INFO] [1676758539.664745]: hello world 1676758539.6646314
[INFO] [1676758539.767539]: hello world 1676758539.7674294
[INFO] [1676758539.864623]: hello world 1676758539.8645182
[INFO] [1676758539.964164]: hello world 1676758539.964054
[INFO] [1676758540.063504]: hello world 1676758540.0633957
[INFO] [1676758540.165181]: hello world 1676758540.1650712
[INFO] [1676758540.263729]: hello world 1676758540.2636213
[INFO] [1676758540.363885]: hello world 1676758540.3637762
[INFO] [1676758540.464499]: hello world 1676758540.464393
[INFO] [1676758540.565477]: hello world 1676758540.5653691
```

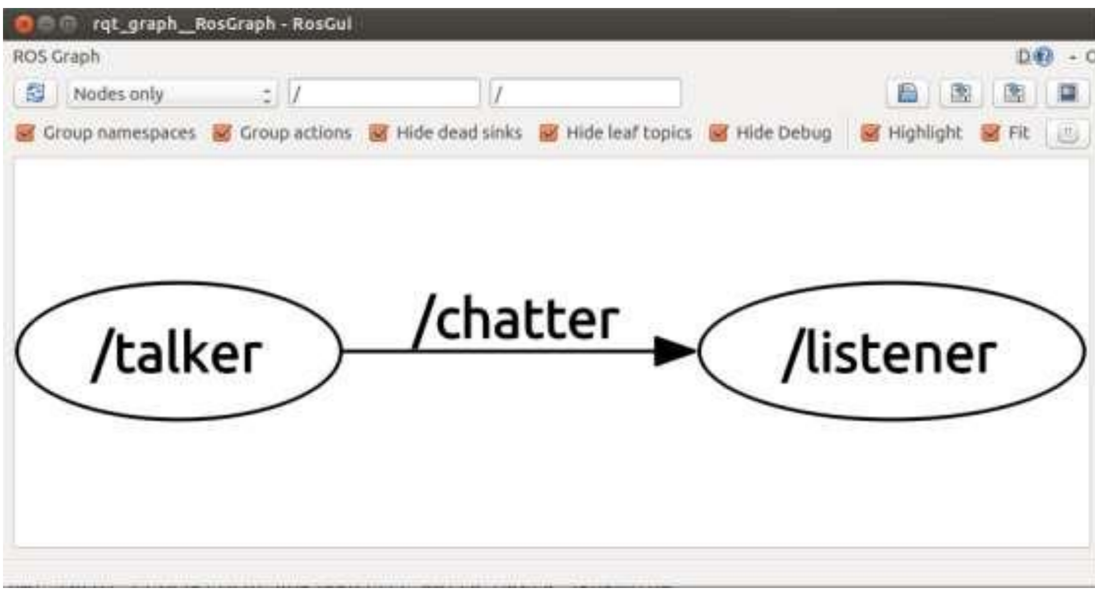
compsci603@compsci603ubuntu2004: ~/catkin_ws

```
[INFO] [1676758538.665498]: /listener_7753_1676758474447I heard hello world 1676758538.664074
[INFO] [1676758538.767663]: /listener_7753_1676758474447I heard hello world 1676758538.7662187
[INFO] [1676758538.865997]: /listener_7753_1676758474447I heard hello world 1676758538.8643656
[INFO] [1676758538.964791]: /listener_7753_1676758474447I heard hello world 1676758538.9634342
[INFO] [1676758539.065998]: /listener_7753_1676758474447I heard hello world 1676758539.0643911
[INFO] [1676758539.167710]: /listener_7753_1676758474447I heard hello world 1676758539.1662862
[INFO] [1676758539.269329]: /listener_7753_1676758474447I heard hello world 1676758539.267621
[INFO] [1676758539.366323]: /listener_7753_1676758474447I heard hello world 1676758539.3648562
[INFO] [1676758539.465543]: /listener_7753_1676758474447I heard hello world 1676758539.4641485
[INFO] [1676758539.565739]: /listener_7753_1676758474447I heard hello world 1676758539.5641482
[INFO] [1676758539.666165]: /listener_7753_1676758474447I heard hello world 1676758539.6646314
[INFO] [1676758539.768974]: /listener_7753_1676758474447I heard hello world 1676758539.7674294
[INFO] [1676758539.865885]: /listener_7753_1676758474447I heard hello world 1676758539.8645182
[INFO] [1676758539.965524]: /listener_7753_1676758474447I heard hello world 1676758539.964054
[INFO] [1676758540.064851]: /listener_7753_1676758474447I heard hello world 1676758540.0633957
[INFO] [1676758540.166399]: /listener_7753_1676758474447I heard hello world 1676758540.1650712
[INFO] [1676758540.265034]: /listener_7753_1676758474447I heard hello world 1676758540.2636213
[INFO] [1676758540.365237]: /listener_7753_1676758474447I heard hello world 1676758540.3637762
[INFO] [1676758540.465797]: /listener_7753_1676758474447I heard hello world 1676758540.464393
[INFO] [1676758540.566665]: /listener_7753_1676758474447I heard hello world 1676758540.5653691
```

ROS Example: Talker and Listener

- To visualize what's going on in ROS, rqt_graph can be used to create a dynamic graph of nodes, topics, etc.

```
$ rosrun rqt_graph rqt_graph
```



ROS Launch

- `roslaunch` is a tool for easily launching multiple ROS nodes as well as setting parameters on the Parameter Server.
- `roslaunch` operates on launch files which are XML files that specify a collection of nodes to launch along with their parameters.
- By convention, these files have a suffix of .launch, with the syntax:

```
$ roslaunch package_name file.launch
```

- `roslaunch` automatically runs roscore.

ROS Launch

- Launch file for launching the talker and listener nodes:

```
<launch>
  <node name="talker" pkg="first_pkg" type="talker.py" output="screen"/>
  <node name="listener" pkg="first_pkg" type="listener.py" output="screen"/>
</launch>
```

- Each <node> tag includes attributes declaring the ROS graph name of the node, the package in which it can be found, and the type of node, which is the filename of the script or executable program.
- output="screen" makes the ROS log messages appear on the launch terminal window.

compsci603@compsci603ubuntu2004: ~/catkin_ws/src/first_pkg/launch

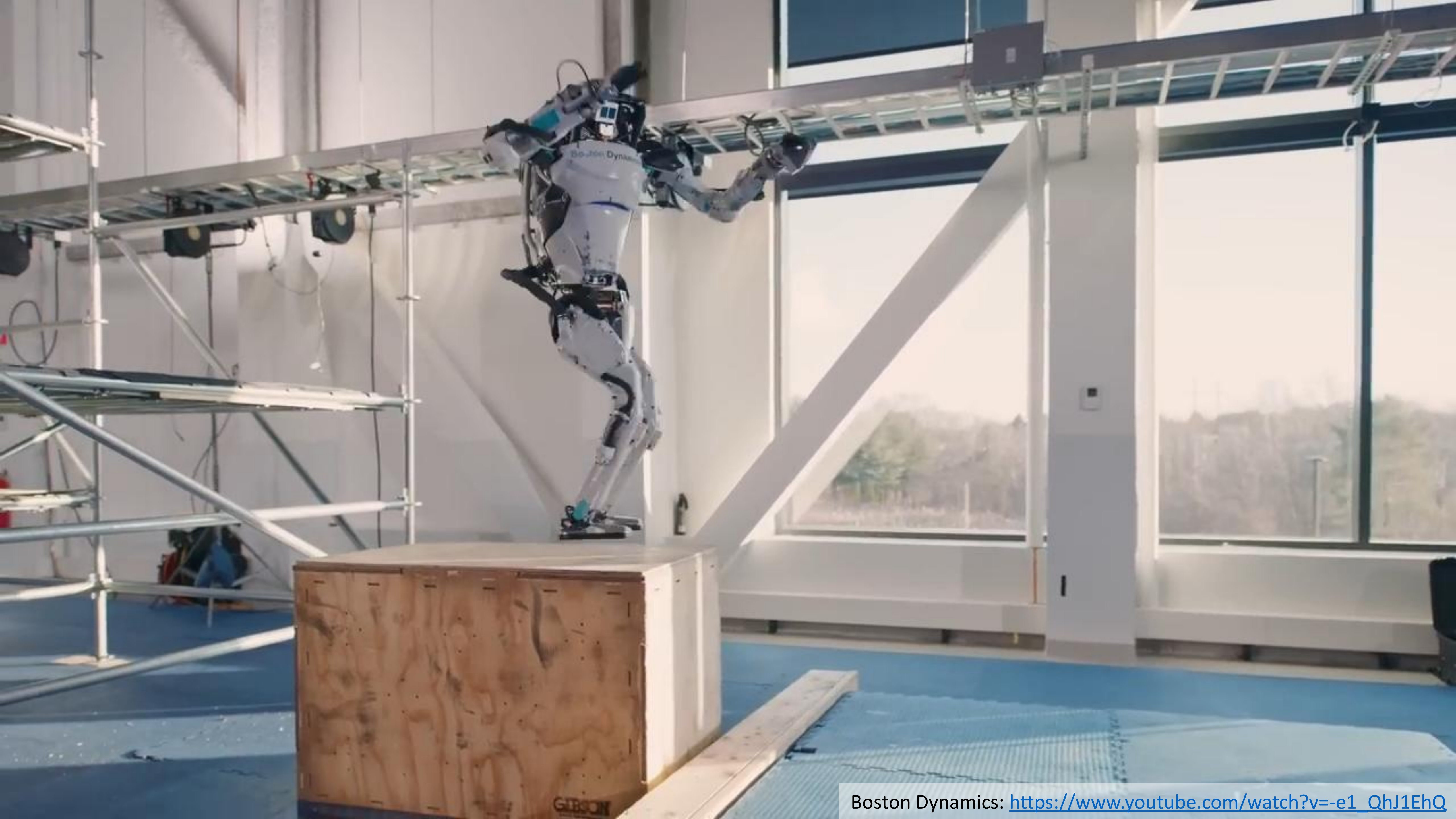
```
compsci603@compsci603ubuntu2004:~/catkin_ws/src/first_pkg/launch$ roslaunch first_pkg first_pkg.launch
... logging to /home/compsci603/.ros/log/adb6f088-afe2-11ed-8d5a-07c22f6a93ff/roslaunch-compsci603ubuntu2004-8642.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://compsci603ubuntu2004:37699/

SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.15

NODES
  /
    listener (first_pkg/listener.py)
    talker (first_pkg/talker.py)

auto-starting new master
process[master]: started with pid [8650]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to adb6f088-afe2-11ed-8d5a-07c22f6a93ff
process[rosout-1]: started with pid [8660]
started core service [/rosout]
process[talker-2]: started with pid [8667]
process[listener-3]: started with pid [8668]
[INFO] [1676762363.759577]: hello world 1676762363.759521
[INFO] [1676762363.862706]: hello world 1676762363.8625958
[INFO] [1676762363.863899]: /listenerI heard hello world 1676762363.8625958
[INFO] [1676762363.961958]: hello world 1676762363.961853
[INFO] [1676762363.963270]: /listenerI heard hello world 1676762363.961853
[INFO] [1676762364.061990]: hello world 1676762364.0618865
[INFO] [1676762364.063235]: /listenerI heard hello world 1676762364.0618865
[INFO] [1676762364.162890]: hello world 1676762364.1627824
[INFO] [1676762364.164855]: /listenerI heard hello world 1676762364.1627824
[INFO] [1676762364.260722]: hello world 1676762364.2606242
[INFO] [1676762364.261943]: /listenerI heard hello world 1676762364.2606242
[INFO] [1676762364.360546]: hello world 1676762364.3604426
[INFO] [1676762364.361886]: /listenerI heard hello world 1676762364.3604426
[INFO] [1676762364.460756]: hello world 1676762364.4606545
[INFO] [1676762364.461968]: /listenerI heard hello world 1676762364.4606545
[INFO] [1676762364.564547]: hello world 1676762364.564437
[INFO] [1676762364.565808]: /listenerI heard hello world 1676762364.564437
[INFO] [1676762364.660302]: hello world 1676762364.6602004
```

ROS Names

- ROS names must be unique.
- If the same node is launched twice, `roscore` directs the older node to exit.
- To change the name of a node on the command line, the special __name remapping syntax can be used.
- The following two shell commands would launch two instances of talker named talker1 and talker2.

```
$ rosrun chat_pkg talker.py __name:=talker1
$ rosrun chat_pkg talker.py __name:=talker2
```

talker1
listener
talker2



Boston Dynamics: https://www.youtube.com/watch?v=-e1_QhJ1EhQ

Atlas failure: https://www.youtube.com/watch?v=aX7KypGlitg