

UTD CSSC PRESENTS:

AN INTRO TO ROBOT OPERATING SYSTEM

ROS

ROS is an important skill for anyone pursuing a
career in robotics.

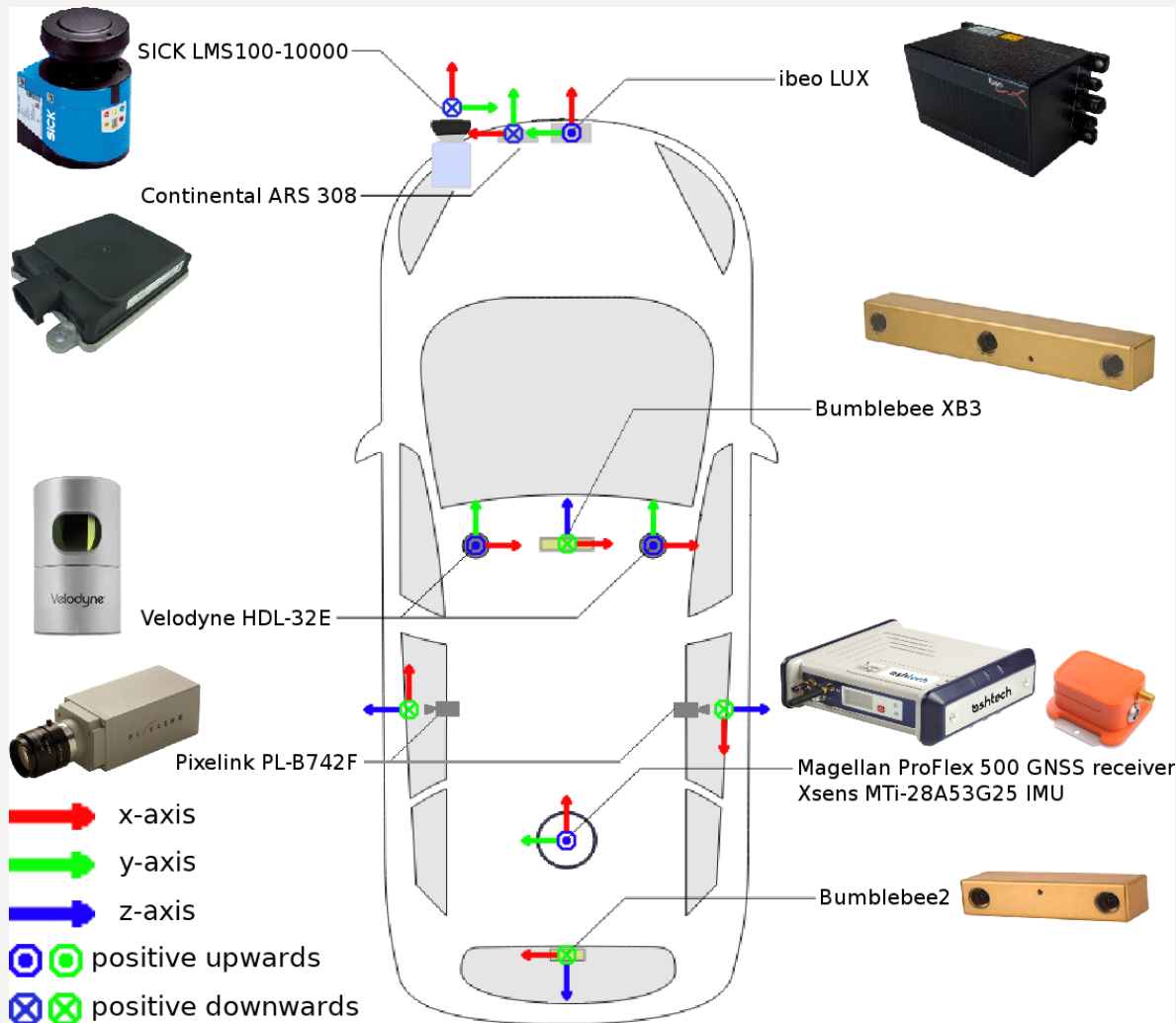
Come learn about what ROS is and how to use it!

Sleiman Safaoui

Sleiman.Safaoui@utdallas.edu

Control, Optimization, and Networks Lab
The University of Texas at Dallas

COMPLEX SYSTEMS



[Z. Yan, et al, "EU long-term dataset with multiple sensors for autonomous driving," 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). 2020.]

WiFi adapter
Standards: IEEE 802.11b/g/n

Wide-angle camera
Logitech C905

Speaker

ZBOX nano XS
AMD E-450 processor 1.65 GHz
2 GB RAM, 64 GB SSD
USB 3.0, HDMI
Gigabit Ethernet
Memory card slot

CM730 control board
Servo communication
3-axis accelerometer
3-axis gyroscope

LiPo battery
14.8 V, 3.6 Ah

20 actuators
Networked Dynamixel
MX actuators
6 per leg (MX-106)
3 per arm (MX-64)
2 in the neck (MX-64)

Lightweight materials
Carbon composite
Aluminum
ABS+

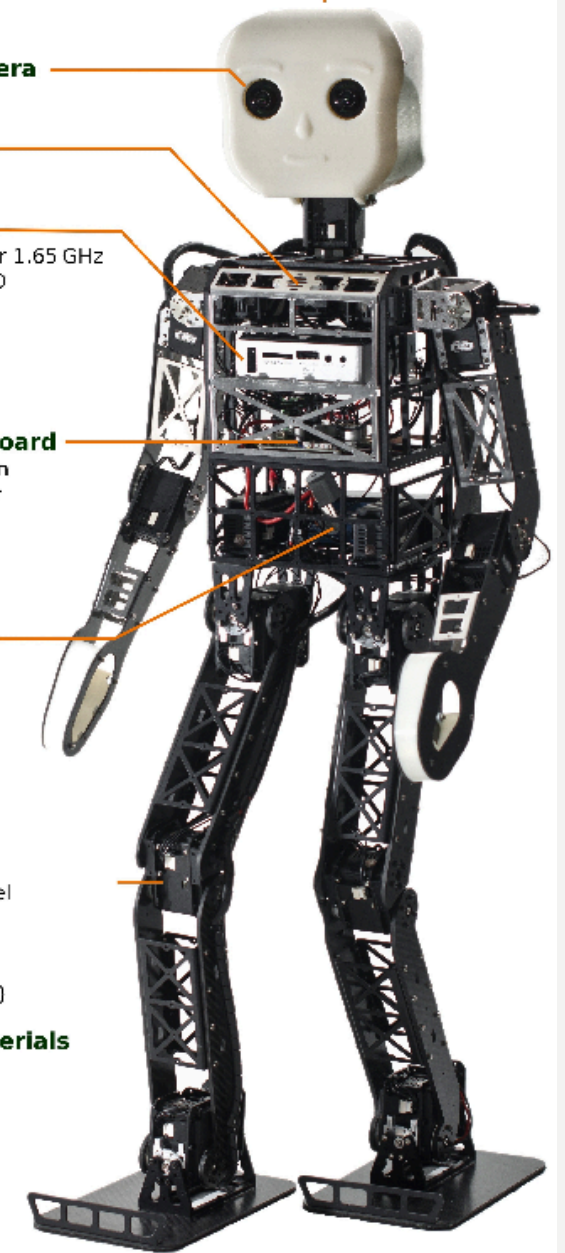
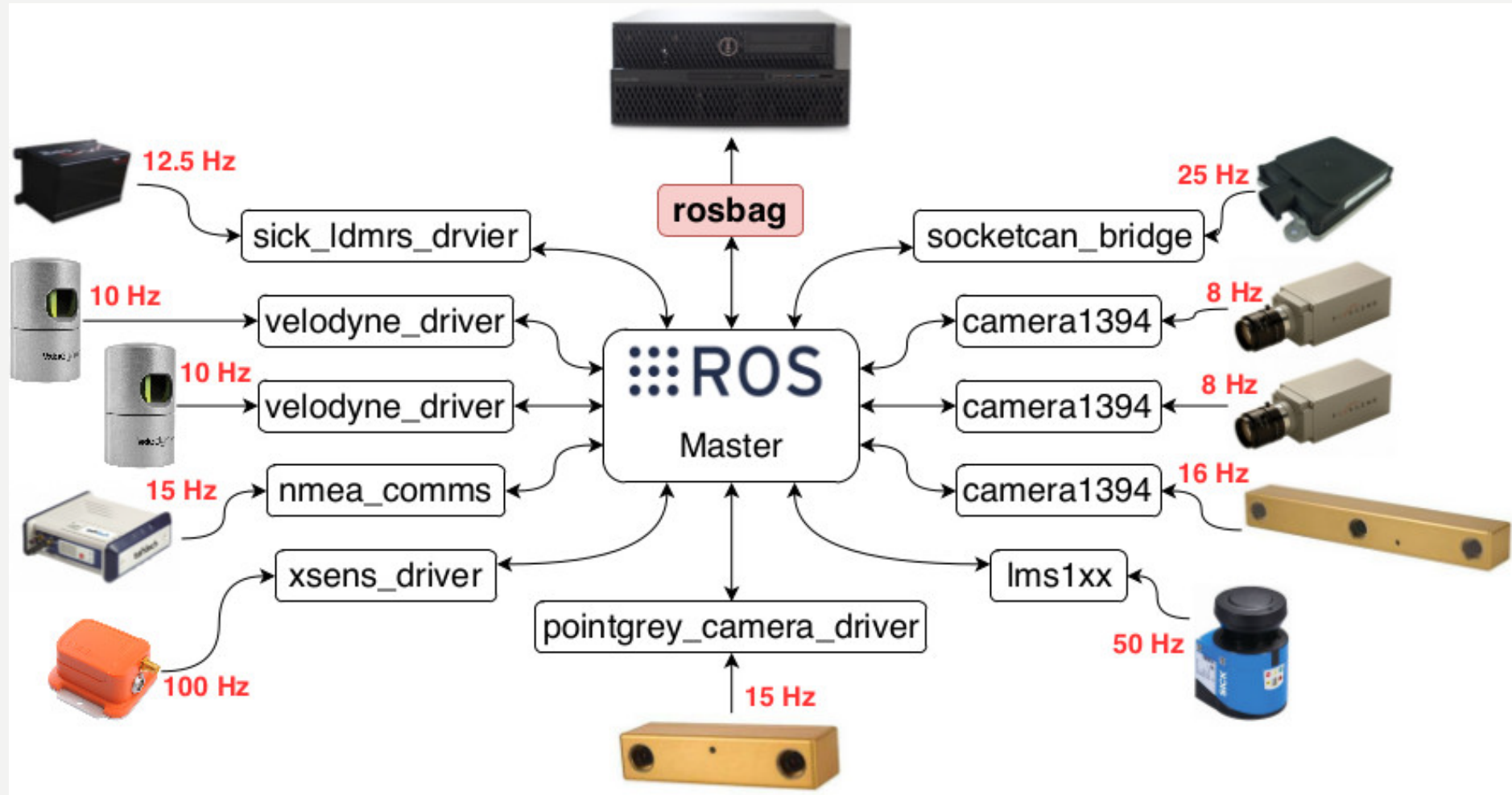


Photo: Felix Oprean

[M. Schwarz, et al, "NimbRo-OP humanoid teensize open platform," IEEE-RAS International Conference on Humanoid Robots, Osaka, 2012.]

COMPLEX INTERACTIONS



[Z. Yan, et al, "EU long-term dataset with multiple sensors for autonomous driving," 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). 2020.]

WHAT IS ROS

- **ROS:Tool**
 - Connects components
 - Sensors: Camera, IMU, Laser scanners ...
 - Actuators: motors, linear actuators, ...
 - Microcontrollers: Arduino, ...
 - Main ROS Components:
 - *Nodes*
 - *Topics*
 - *Publish*
 - *Subscribe*

WHAT IS ROS

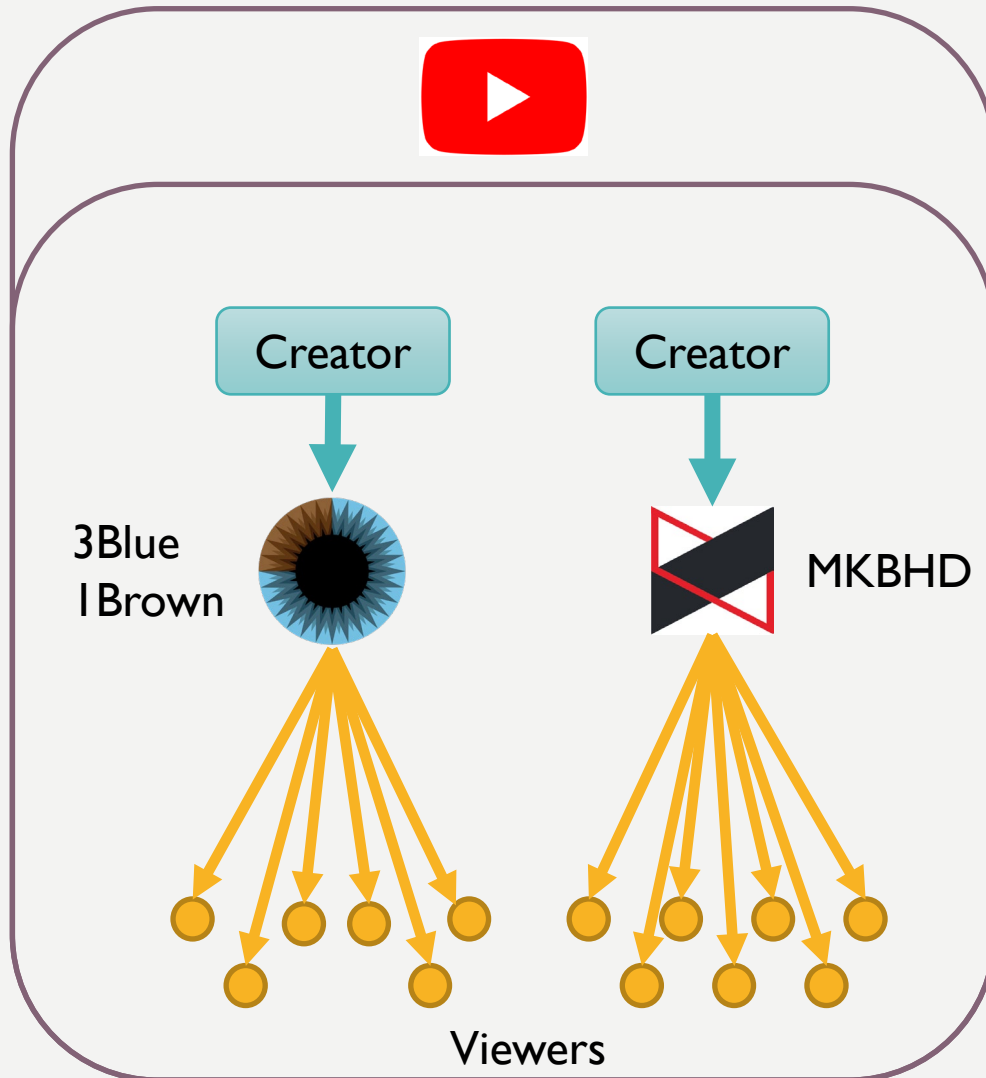
- **ROS:Tool**

- Connects components
 - Sensors: Camera, IMU, Laser scanners ...
 - Actuators: motors, linear actuators, ...
 - Microcontrollers: Arduino, ...
- Main ROS Components:
 - *Nodes*
 - *Topics*
 - *Publish*
 - *Subscribe*

- **ROS: Community**

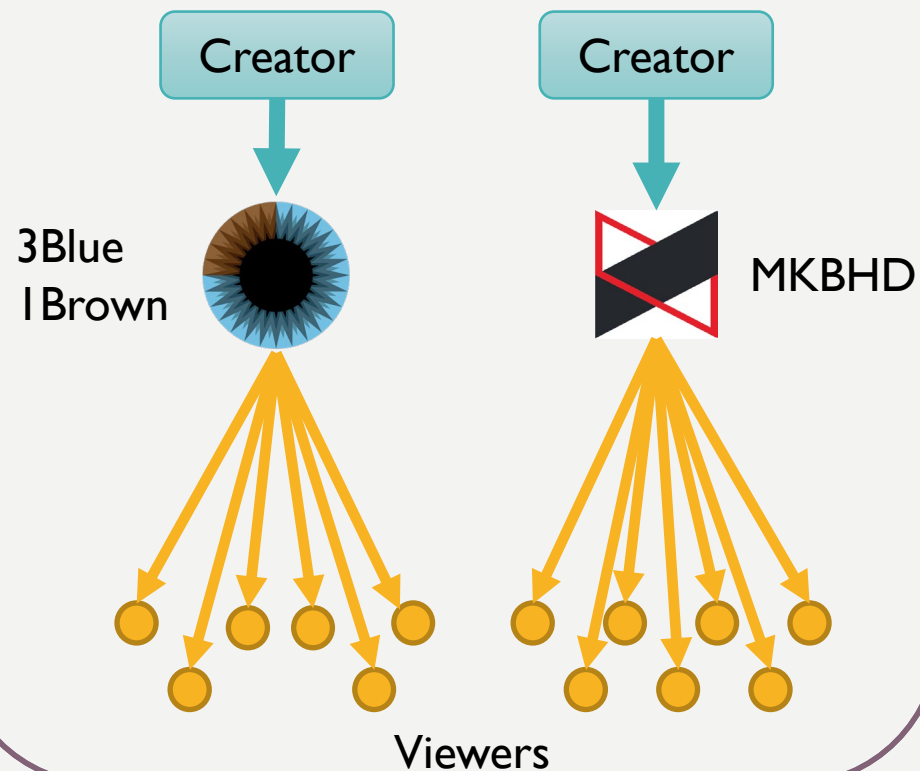
- Many open-source software packages
 - Data processing: camera (OpenCV), lidar (PCL), ...
 - Arm navigation (MoveIt)
 - Mapping and Localization (gmapping, amcl, ...)
 - Simulation (Gazebo)
- Forums
- Tutorials
- [600+ companies use ROS](#)

PUB/SUB MODEL: ANALOGY

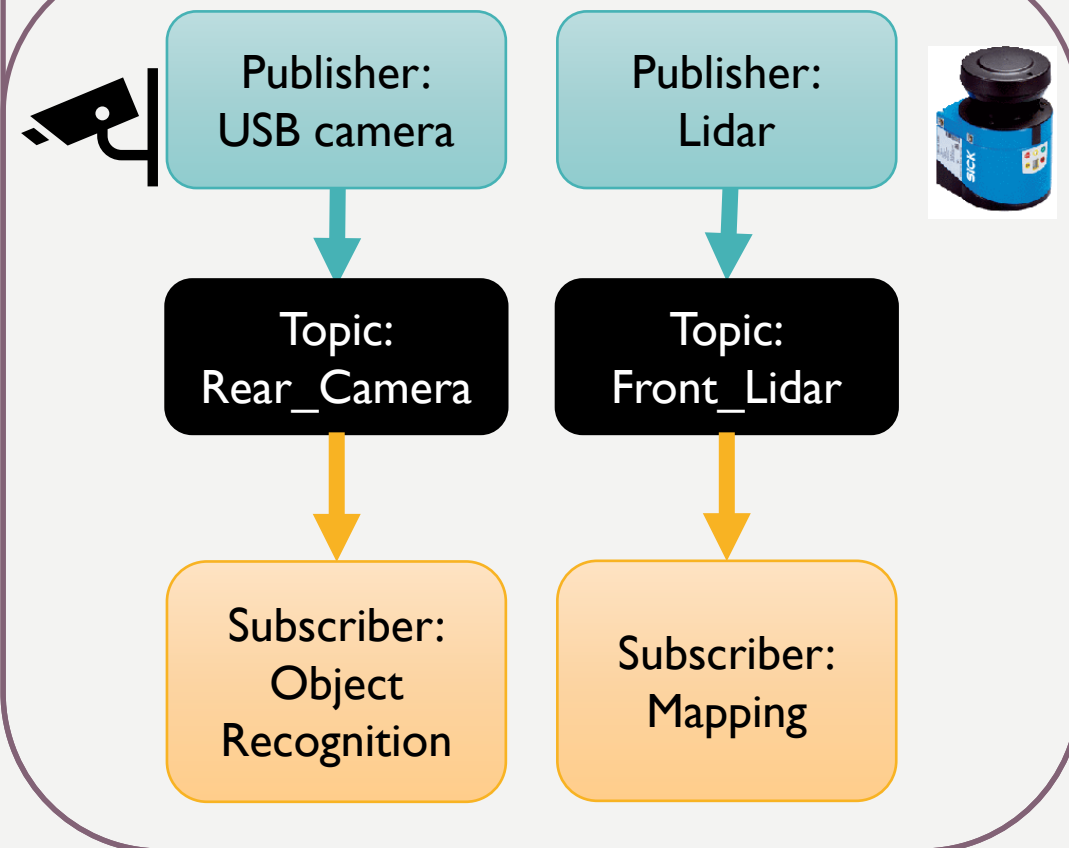


- YouTube:
 - Channels provide content
 - Creators **publish** content
 - regardless of viewership
 - Viewers **subscribe** to the channel
 - regardless of videos being published
- Topic = Channel
- Publish = send data to topic
- Subscribe = receive/wait for data from a topic

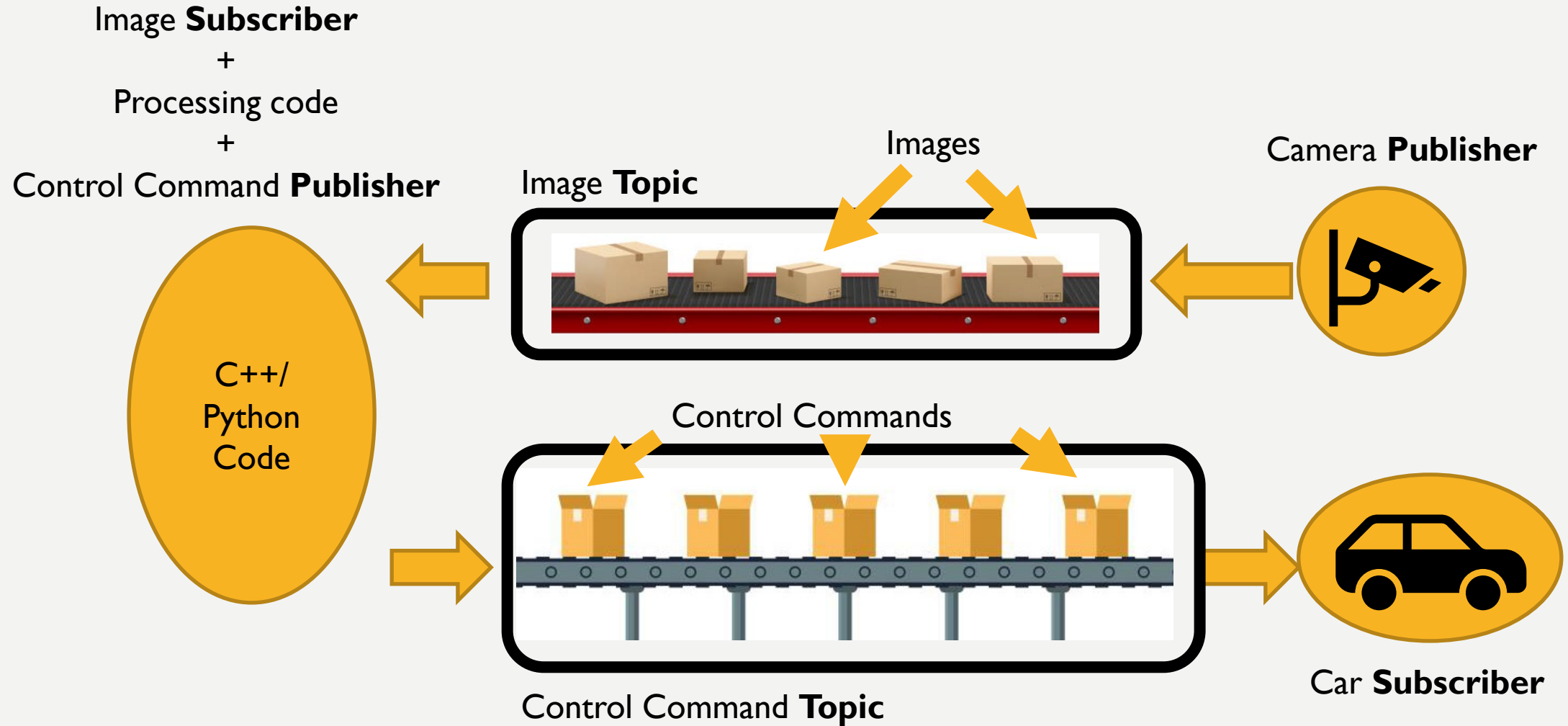
PUB/SUB MODEL



 **ROS** ROS Master

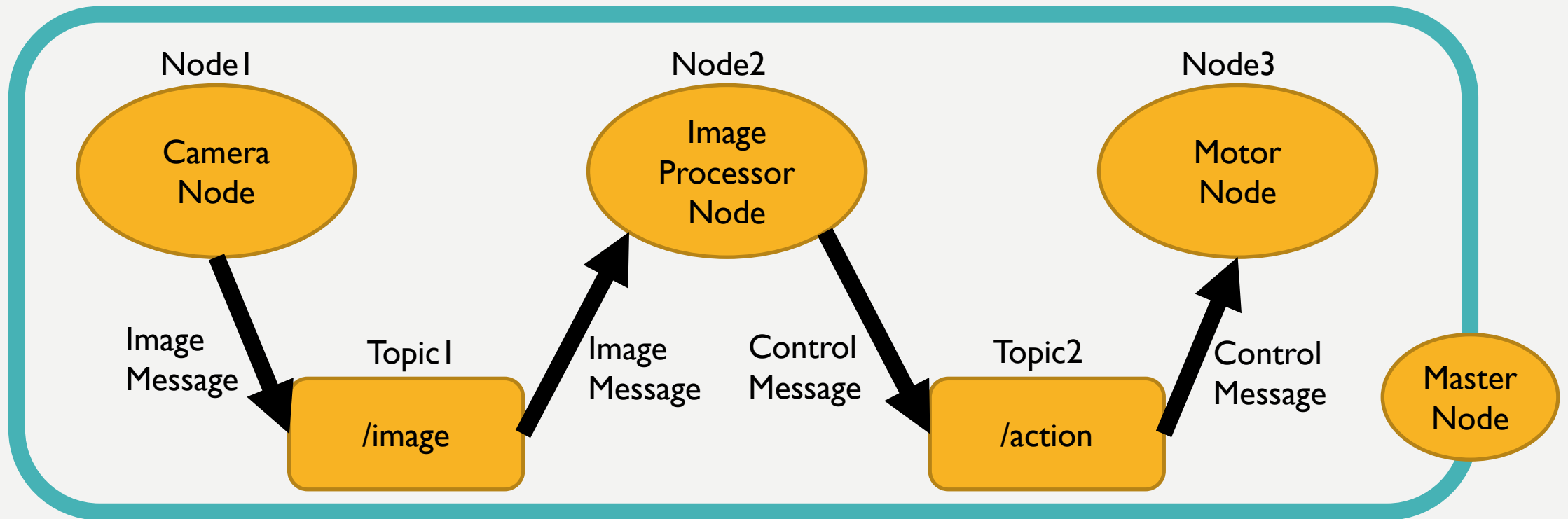


EXAMPLE



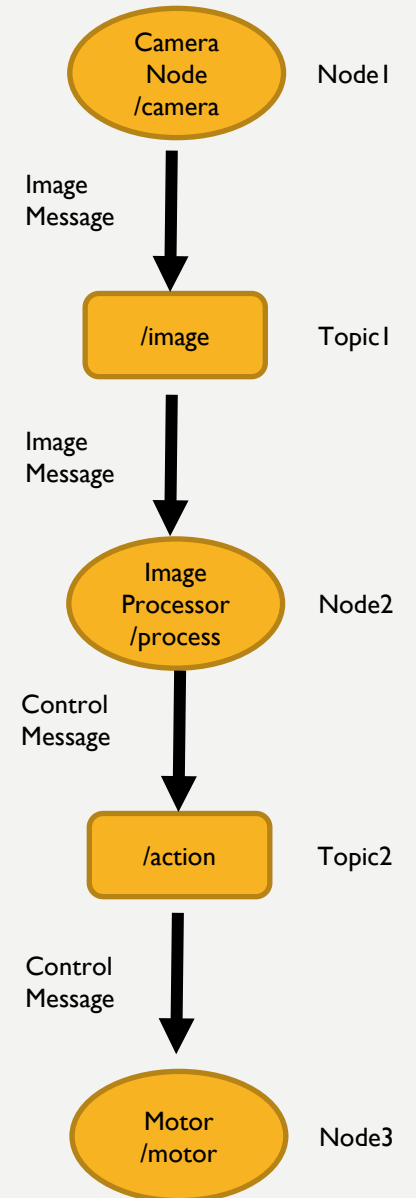
NODES, TOPICS, MESSAGES

- Node: Processing unit – does some computation (Typically C++ or Python)
- Nodes talk over Topics
 - Send (publish) messages, Receive (subscribe) message, or Both



ROS TOOLS: NODES, TOPICS, MESSAGES

- [rosc](#) (require master)
 - `rosc list`: lists all nodes
 - `rosc info <node>`: gives information about a node
- [rostopic](#) (require master)
 - `rostopic list`: lists all topics
 - `rostopic info <topic>`: gives information about the topic
 - `rostopic echo <topic>`: displays the data through a topic
 - `rostopic hz <topic>`: finds the frequency of publishing to a topic
- [rosmg](#)
 - `rosmg list`: lists all messages
 - `rosmg show <msgType>`: display message type details

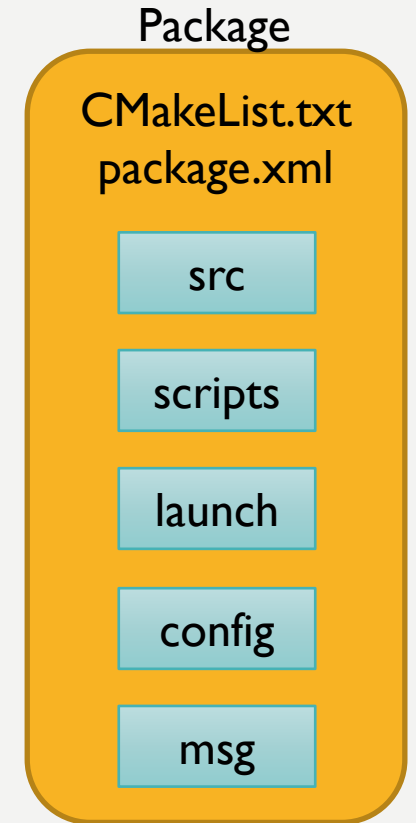


USING ROS

- ROS is *thin*: Integrates into code
- Integration in C++/Python
 1. Create a script (C++, Python, ...)
 2. Import the ROS library (roscpp for C++, rospy for Python, ...)
 3. Define a node
 - e.g. in Python: `rospy.init_node('process')`
 4. Define publishers or subscribers (or both)
 - e.g. in Python: `pub = rospy.Publisher('/action', control_msg)`
 5. Publish or subscribe
 - e.g. in Python: `pub.publish(data)`
- Run the script with `roslaunch`
 - e.g. `roslaunch image_package process_image.py`

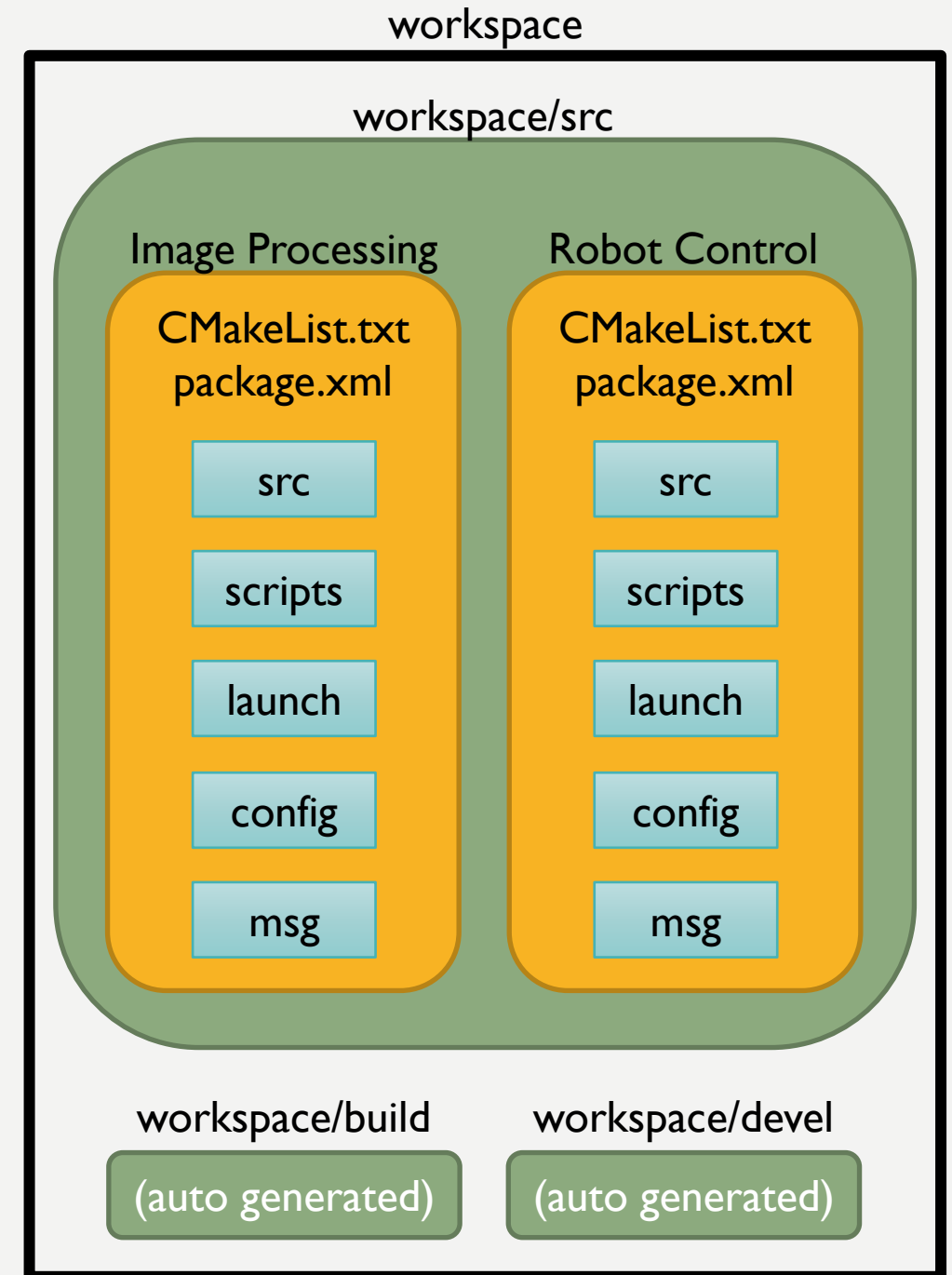
ROS PACKAGE

- **Package**
 - **CMakeLists.txt** (defines dependencies and requirements)
 - **package.xml** (defines package name, dependencies, ...)
 - **src** (files that define nodes (.cpp, .py))
 - **scripts** (files that don't define nodes)
 - **launch** (.launch files to automatically run many nodes)
 - **config** (YAML files with parameters for users to edit)
 - **msg** (contains custom message definitions: .msg files)
- **Examples:**
 - Image processing package: process images in different ways
 - Mapping package: generate a map of the environment
 - Planning package: plan the robot path



CATKIN

- We have many components:
 - C++ files
 - Python scripts
 - Custom Messages
 - Multiple packages
- What to do?
- **Catkin Workspace:** directory for all ROS development
- **catkin_make:** tool that helps compile and link all the different components



WORKSPACE OVERVIEW

- **workspace** (use `catkin_make` in this directory)
 - *build* (auto generated)
 - *devel* (auto generated)
 - `setup.bash` (source this so ROS can “see” all files: `source workspace/devel/setup.bash`))
 - **src** (all the code you write is here)
 - **ImageProcessingPackage**
 - CMakeLists.txt, package.xml
 - src
 - `detect_people.py`, `find_traffic_signs.cpp`, ...
 - **MappingPackage**
 - CMakeLists.txt, package.xml
 - src
 - **ControlPackage**
 - CMakeLists.txt, package.xml
 - src

ROSRUN & ROSLAUNCH

The code is written.

The package is built.

How do you run the code with ROS?

ROSRUN & ROSLAUNCH

- `roslaunch`
 - Starts a single node
- `roslaunch`
 - 1. Start the ROS master (if not already started)
 - run in a terminal: `roscore`
 - 2. Start the file
 - `roslaunch <package_name> <node_name>.<cpp, py, ...>`

ROSRUN & ROSLAUNCH

- **roslaunch**
 - Starts many nodes, possibly with certain parameters, and starts the master (if needed)
 1. Write a .launch script with all nodes
 2. Start the script
 - `roslaunch <package name> <launch file name>.launch`

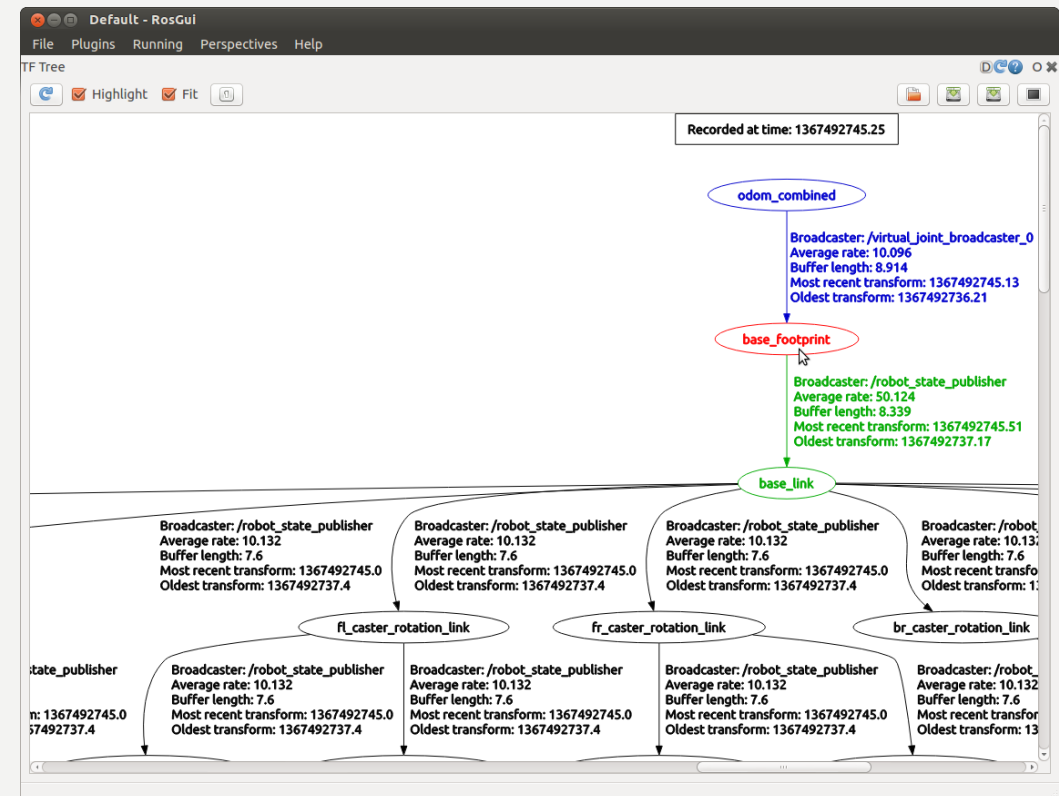
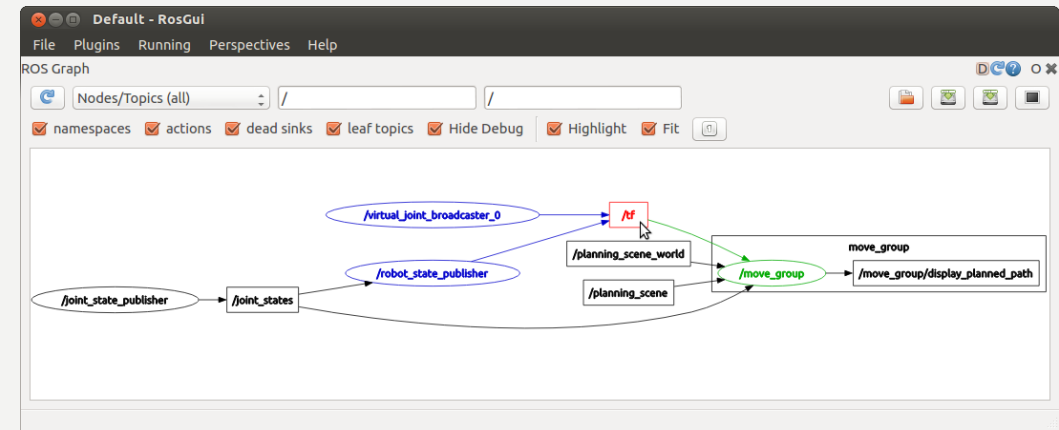
PARAMETER SERVER

- Parameters needed (across many scripts)
- ROS parameter server
 - Tracks all parameters
 - Managed by ROS master
 - Any node can get and set a parameter value
 - Running nodes not affected
 - Examples
 - /move_base/local_costmap/height
 - /usb_cam/framerate
 - /gazebo/time_step

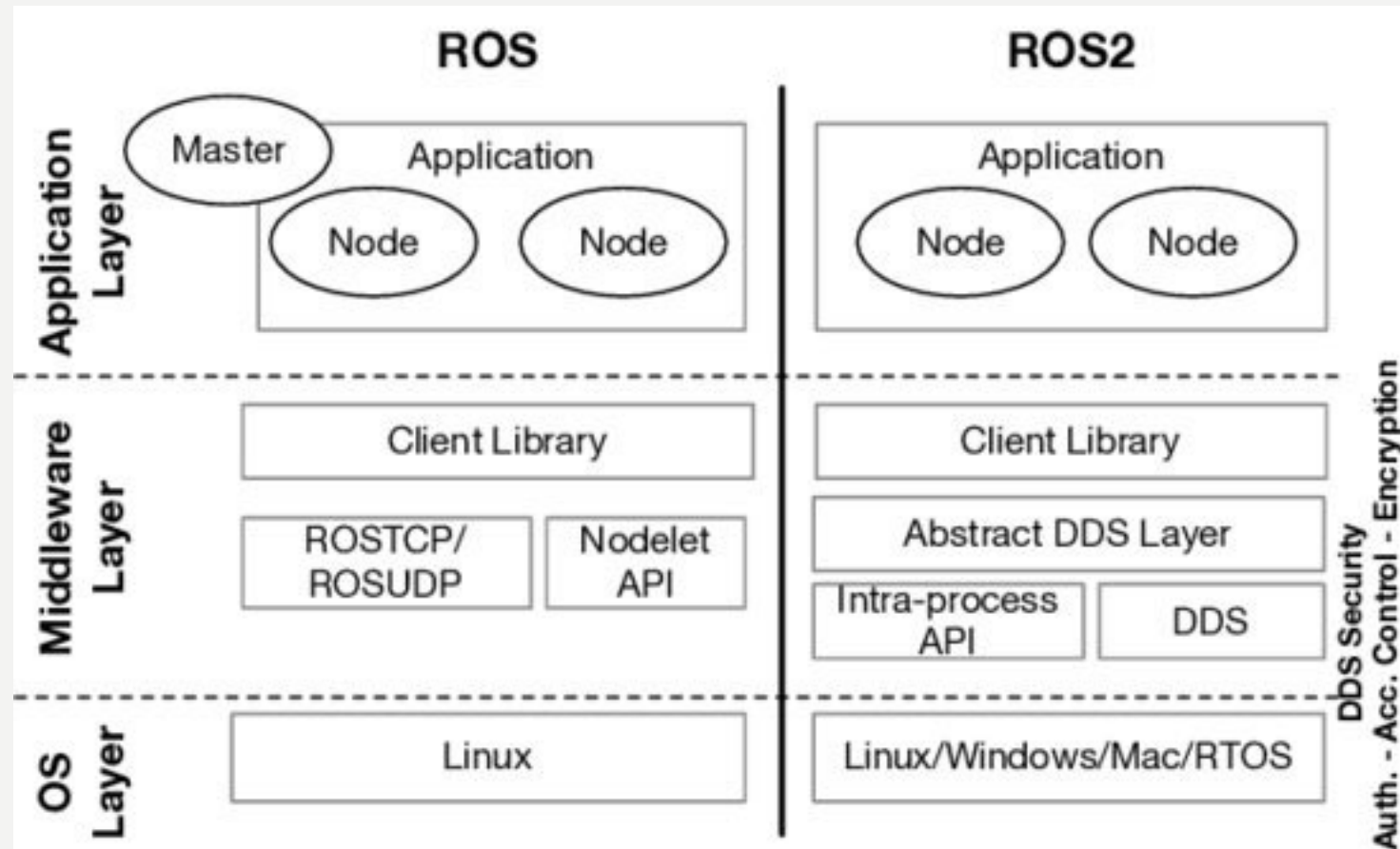
DEBUGGING

- Many ROS tools

- rqt_console
- rqt_graph
- rqt_tf_tree
- rosbag
 - rosbag record <topic>
 - rosbag play file.bag
- rospy.loginfo()



ROS1 AND ROS2



[G. Mazzeo and S. Mariacarla, "TROS: Protecting Humanoids ROS from Privileged Attackers, " International Journal of Social Robotics, Vol. 12, pp 827-841, 2020.]

ROS is more mature than ROS2
Final ROS distro: Noetic (2020)

ROS2 distros:
Foxy (2020), Humble (2022)

SUBMITTED QUESTIONS

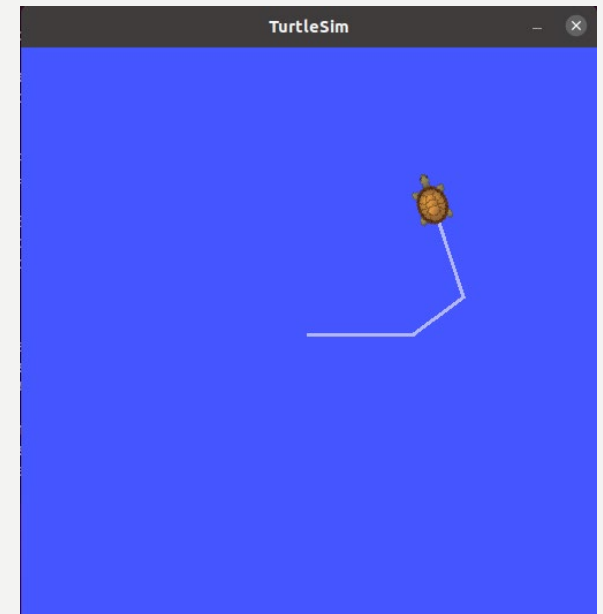
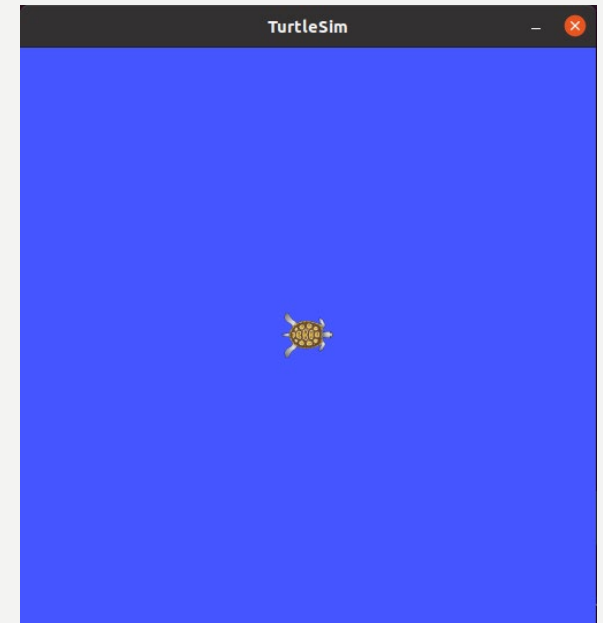
- Robots and microcontrollers
 - Arduino boards are common
 - Interface with them via roserial package
- Integrating multiple sensors
 - Publish sensor data to various topics
 - Write scripts to subscribe, process the data, and publish processed info
 - Write a script to subscribe to the processed data and do something
- Robotic Prosthetics
 - Humanoid robots (e.g. PR2 robot), robot manipulators, ... very common
 - Shadow Robot (prosthetics company) uses ROS
- Component control
 - Many ROS tools allow you to design the controller then interface with an actuator (e.g. via an Arduino or ESC)



DEMO

TURTLESIM DEMO

- Turtlesim is a package that comes preinstalled with the ROS full installation.
- Package name: `turtlesim`
- Some ROS nodes:
 - `turtlesim_node`: starts a turtle simulator
 - `turtle_teleop_key`: moves the turtle using the keyboard arrow keys



TURTLESIM DEMO

- All text in code-font should be executed in a terminal.
- If a terminal window is in-use, open another terminal window.
- Start roscore (new terminal)
 - roscore
- Start the turtlesim node (new terminal)
 - rosrun turtlesim turtlesim_node
- See nodes and topics (new terminal)
 - rosnode list
 - rostopic list
 - rqt_graph

```
demo@demo:~$ rosnode list
/rosout
/turtlesim
demo@demo:~$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

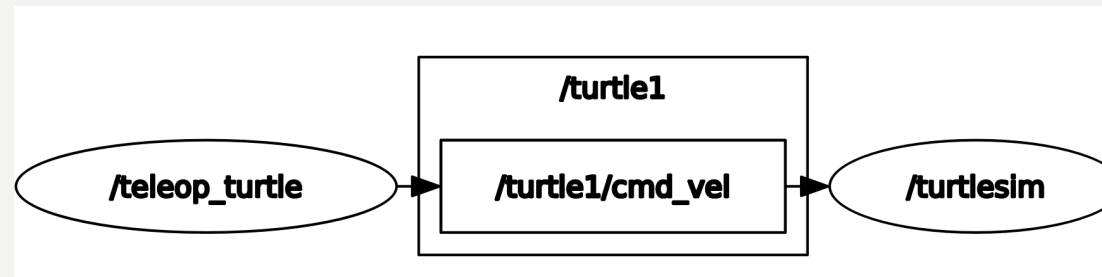
/turtlesim

TURTLESIM DEMO

- **Start the teleoperator node (new terminal)**
 - `roslaunch turtlesim turtlesim_teleop_key`
- **See nodes and topics (in a new terminal)**
 - `rostopic list`
 - `rostopic echo /turtle1/cmd_vel`

```
demo@demo:~$ rosnodetop
rosnodelist
-----
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose_
```

```
demo@demo:~$ rostopic list
/rosout
/turtlesim
demo@demo:~$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```



/turtlesim

TUTORIAL: CREATE WORKSPACE

- Creating a catkin workspace:
 - `cd` # goes to the home directory (~)
 - `mkdir -p workspace/src` # creates (~/workspace/src)
 - `cd workspace` # goes to ~/workspace
 - `catkin_make` # builds the ROS workspace
- You should now have `build`, `devel` **folders** in workspace
- Source your workspace (every time you start a new terminal)
 - `source ~/workspace/devel/setup.bash`

TUTORIAL: CREATE PACKAGE

- You can create a package manually or using `catkin_create_package`
 - `cd ~/workspace/src` # go to ~/workspace/src, all packages go here
 - `catkin_create_pkg demo_package std_msgs rospy`
- You now have an empty package called `demo_package` with dependencies `std_msgs`, `rospy`
- Build the workspace
 - `cd ~/workspace`
 - `catkin_make` # compiles the ROS workspace

TUTORIAL: CREATE A NODE

- Go to the package `src` folder (`~/workspace/src/demo_package/src`) and create a script. Let's call it `publisher_node.py`
- [Add code.](#)
- This is a python script, so
 - we need to make it an executable: `chmod +x publisher_node.py`
 - we do not need to rerun `catkin_make`
- Start the ROS master
 - `roscore`
- Run the script
 - `roslaunch demo_package publisher_node.py`
- Analyze the script
 - E.g. `rostopic echo /counter`

TUTORIAL: MULTIPLE PACKAGES

- In [workspace2](#), two packages are provided:
 - `data_publisher`: mimics a sensor publishing distance data and the associated confidence (probability) with the distance
 - `data_analyzer`: subscribes to the distance and probability data, analyzes them, and publishes a message about the safety status (safe, distance too small, low confidence)
- Run the two scripts `publisher_node.py` and `safet_based_on_data.py` and analyze what is going on using `rostopic list`, `rostopic echo <topic name>`, `rqt_graph`, ...
 - Hint 1: Review the slide on [roslaunch](#) slide (slide 17)
 - Hint 2: Review how we used `roslaunch` in previous slides
 - Hint 3: See next slide for commands

TUTORIAL: MULTIPLE PACKAGES

- In [workspace2](#), two packages are provided:
 - `data_publisher`: mimics a sensor publishing distance data and the associated confidence (probability) with the distance
 - `data_analyzer`: subscribes to the distance and probability data, analyzes them, and publishes a message about the safety status (safe, distance too small, low confidence)
- Run the two scripts `publisher_node.py` and `safet_based_on_data.py` and analyze what is going on using `rostopic list`, `rostopic echo <topic name>`, `rqt_graph`, ...
 - Run [roslaunch](#) `data_publisher publisher_node.py`
 - Run (new terminal) `roslaunch data_analyzer safet_based_on_data.py`
 - Run (new terminal) `rostopic echo \data`
 - Run (new terminal) `rostopic echo \decision`

DEBUGGING TIPS

- Python not finding ROS package (e.g. when importing custom messages)
 - Make sure the package has been compiled
 - Make sure your custom message is defined properly in `CMakeLists.txt`
 - Make sure message generation and message runtime dependencies are defined in `CMakeLists.txt` and `package.xml`
 - Make sure your `workspace/devel/setup.bash` is sourced
 - Make sure ROS master is running
- `rostopic echo <topic>` **throws** `ERROR: Cannot load message class for [<message_type>]`. Are your messages built?
 - Make sure your `workspace/devel/setup.bash` is sourced
 - Make sure ROS master is running
 - Make sure your message shows up using [`rosmmsg show`](#)
- Check forums, ROS wiki, or Google the errors