Introduction to ROS2: Basics, Motion, and Vision

Geesara Kulathunga

INNOPOLIS UNIVERSITY

Content

- Working with tmux
- Workspace
- Build System
- ROS1 vs ROS2
- Nodes
- ROS1 Nodelets and ROS2 Composition
- Topics and Messages
- Launch and run
- Package
- Publisher, Subscriber, and Service
- Parameters

Working with tmux

tmux, a program that runs in a terminal. It allows multiple other terminal programs to be run inside it. To install tmux: **sudo apt install tmux**

`Ctrl+b c`	Create a new window (with shell)
`Ctrl+b w`	Choose window from a list
`Ctrl+b 0`	Switch to window 0 (by number)
`Ctrl+b ,`	Rename the current window
`Ctrl+b %`	Split current pane horizontally into two panes
`Ctrl+b "`	Split current pane vertically into two panes
`Ctrl+b o`	Go to the next pane
`Ctrl+b ;`	Toggle between the current and previous pane
`Ctrl+b x`	Close the current pane

ROS Workspace

Check available ROS2 packages

apt-cache search ros_version, e.g., ros-humble

Set PATH and enable ROS2 within the system

```
source /opt/ros/ros_version/setup.bash
echo "source /opt/ros/ros_version/setup.bash" »  /.bashrc
```

change /opt/ros/ in case if you installed into another location

ROS Workspace

1. Default workspace is located at /some_path/ros/ros_version/setup.bash

You can create ros workspace in a location you prefer

```
mkdir -p  /catkin_ws/src
cd  /catkin_ws
colcon build
cd ./install && pwd
echo source ‘pwd‘/setup.bash »  /.bashrc
source install/setup.bash
echo $AMENT_PREFIX_PATH
```

ROS Build System

1. **colcon build** is used to build the the ros packages and generate executable, libraries, and interfaces

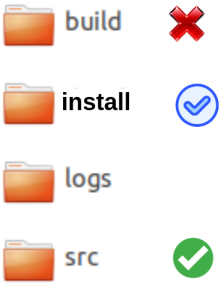
to navigate to workspace

```
>cd ~/catkin_ws
```

to build your package

```
>colcon build --packages-select package_name
Note:whenever package is built, it is required to
>source ./install/setup.bash
```

ROS Build System



to see catkin workspace

colcon list

Example

Hello world!

```
> cd  /catkin_ws/src
> git clone https://github.com/GPrathap/ros2_intro.git
> cd ../ && colcon build --packages-select hello_world
> source install/setup.bash
> ros2 launch hello_world pub_sub_variant_1.py
```

ROS1 Master (roscore) No more in ROS2

Figure: https://www.youtube.com/watch?v=NmidmSS9Ylk

ROS1 Master (roscore)

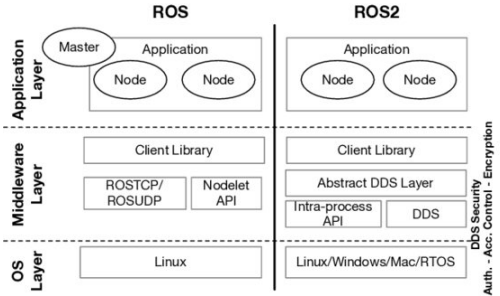
1. The centralized controller or manager
2. Register nodes (sub-programs) when starts with the master
3. Handle communication between nodes (sub-programs (nodes))
4. Also, provide the Parameter server, which is shared among the Nodes that is used to retrieve parameters
5. rosout, which is /rosout, logging purpose
6. roscore = master + parameter_server + rosout

to start the master

```
roscore
```

ROS2 there is no master, no parameter server, hence no roscore. All parameters are node-specific

ROS1 vs ROS2



DDS (Data Distribution Service) is an open-standard connectivity framework for real-time systems, which enables distributed systems to operate securely as an integrated whole.

Mazzeo, G., Staffa, M. (2020). TROS: Protecting Humanoids ROS from Privileged Attackers. International Journal of Social Robotics, 12, 827-841.

ROS Nodes

Figure: https://www.youtube.com/watch?v=NmidmSS9Ylk

ROS Nodes (Processors/pub-programs)

1. Do you know the different between threads and processors
2. Each nodes executes as a processor
3. Node APIs: rclcpp, rclpy

to run a node

```
ros2 run package_name node_name
```

to see active nodes

```
ros2 node list
```

to get information about a node

```
ros2 node info node_name
```

ROS Topics

Figure: https://www.youtube.com/watch?v=NmidmSS9Ylk

ROS Topics

1. Topics can be used to communicate among the nodes
2. Nodes can publish, subscribe or both, typically 1 to n connection exist between a publisher and subscribers

to see active topics list

```
ros2 topic list
```

to subscribe to a topic

```
ros2 topic echo /topic
```

to get information about a topic

```
ros2 topic info topic_name
```

ROS Launch

1. The configuration of the system includes what programs to run, where to run them, what arguments to pass them
2. Launch files are written in Python, XML, or YAML
3. **executable**: executable of the node
4. **package**: which package that the considered node belongs
5. **parameters**: parameters to be passed to the node
6. **output**: where to log the output: console or log file

ROS Package

1. **launch folder**: contains launch files each of which may have defined multiple nodes or includes another multiple launch files
2. **src folder**: source files
3. **package.xml**: or manifest file, contains the package meta data
4. **CMakeLists.txt**: dependencies, executable, and exporting all meta information

ROS Package Creation

dummy package with several dependencies

ros2 pkg create <package_name> −−dependencies [depend1] [depend2] [depend3]

```
> ros2 pkg create first_package −−dependencies rclcpp std_msgs
> source install/setup.bash
```

ROS Package's Package.xml

1. **name**: name of the package
2. **version**: it should be defined with three integers separated by dots
3. **description**: objective of the package
4. **buildtool_depend**: dependencies that are required for the build tool
5. **build_depend**: dependencies of the package
6. **build_export_depend**: dependencies that are included in the headers
7. **exec_depend**: dependencies of shared libraries

ROS Messages

ROS Messages

ROS Messages

1. Message contains information to be transformed
2. Typically comprises of a nested structure of primitive data types, e.g., integer, double, float, boolean, and string.
3. Define as *.msg

to see type of a topic

```
ros2 topic type /topic
```

to publish a message over a topic

```
ros2 topic pub /topic type <message>
```

ROS Messages

Odometry message example

```
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

Header message example

```
uint32 seq
time stamp
string frame_id
```

More info: http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html

ROS Message Create

Friend's message

mkdir -p catkin_ws/src/hello_world/msg
cd catkin_ws/src/hello_world/msg
touch **R**obot**D**etails.msg

Robot's message content

string name
string id

ROS Message: Standard Types

Primitive type	Serialization	C++	Python
bool (1)	unsigned 8-bit int	uint8_t(2)	bool
int8	signed 8-bit int	int8_t	int
uint8	unsigned 8-bit int	uint8_t	int(3)
int16	signed 16-bit int	int16_t	int
uint16	unsigned 16-bit int	uint16_t	int
int32	signed 32-bit int	int32_t	int
uint32	unsigned 32-bit int	uint32_t	int
int64	signed 64-bit int	int64_t	long
uint64	unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string (4)	std::string	string
time	secs/nsecs signed 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

ROS Message Create Cont.

Package Dependencies

buildtool_depend: ament_cmake
depend: rclcpp
builtin_interfaces
rosidl_default_generators
action_msgs

exec_depend: rosidl_default_runtime

ROS Message Create Cont.

to find dependencies

```
find_package(ament_cmake REQUIRED)
find_package(builtin_interfaces REQUIRED)
find_package(rosidl_default_generators REQUIRED)
```

to generate messages

```
rosidl_generate_interfaces(PROJECT_NAME
"msg/FriendInfo.msg"
"msg/R2D2.msg"
"srv/FriendInfoService.srv"
DEPENDENCIES builtin_interfaces )
```

ROS Service

ROS Service

1. Peer-to-peer
2. Execute sequentially, i.e, request and then have to wait till the response
3. *.srv is the file type that defines the service that has two parts: a request and a response. When creating a service, request and response is separated by "—", which is given in the next slide
4. Similar analogy how topic works, yet services are two way transports. A service does one-to-one communication, topic does many to many communication

ROS Service Create

create a service

```
mkdir -p catkin_ws/src/hello_world/srv
cd catkin_ws/src/hello_world/srv
touch RobotActions.srv
```

service

```
string name
string id
---
string heartbeat
```

ROS Service Create Cont.

to generate a service (define in CMakeLists.txt)

```
rosidl_generate_interfaces(PROJECT_NAME
"msg/FriendInfo.msg"
"msg/R2D2.msg"
"srv/FriendInfoService.srv"
DEPENDENCIES builtin_interfaces )
```

ROS Service

to run a service

ros2 launch hello_world service_server.py

to call a service

ros2 launch hello_world service_client.py
ros2 service call service_name message
ros2 service call /set_heartbeat friend_msgs/srv/FriendInfoService "id: 456, value: 5678"
ros2 run hello_world service_client 4567

ROS Publisher

to write a publisher

```
mkdir -p catkin_ws/src/first_package/scripts
cd catkin_ws/src/first_package/scripts
touch hello_pub.py
chmod +x    hello_pub.py
```

ROS Publisher

ROS Publisher Create

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.callback)
        self.i = 0
    def callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % 567
        self.publisher_.publish(msg)
```

ROS Subscriber

ROS Subscriber

to write a subscriber

```
mkdir -p catkin_ws/src/first_package/scripts
cd catkin_ws/src/first_package/scripts
touch hello_sub.py
chmod +x    hello_sub.py
```

ROS Subscriber Create

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,'topic', self.listener_callback, 10)
        self.subscription  # prevent unused variable warning
    def listener_callback(self, msg):
        self.get_logger().info('I␣heard:␣"%s"' % msg.data)
```

ROS Subscriber and Publisher Install

to install scripts (define in CMakeLists.txt)

```
install(PROGRAMS scripts/hello_pub.py scripts/hello_sub.py
DESTINATION lib/$PROJECT_NAME)
```

Let’s try to say hello!

to run

```
ros2 run hello_world hello_sub.py
ros2 run hello_world hello_pub.py
```

ROS parameters

Parameters in ROS are associated with **individual nodes**. Parameters are used to **configure** nodes **at startup (and during runtime)**, without changing the code. The **lifetime** of a parameter is tied to the **lifetime of the node**.

- To see the parameters that belongs to each node **ros2 param list**
- To get a parameter value **ros2 param get /node_name param_name**
- To set a parameter value **ros2 param set /node_name param_name value**
- To load a set of parameters **ros2 param load /node_name param_list.yaml**
- To load a set of parameters when it starts **ros2 run /package_name /node_name −−ros-args −−params-file param_list.yaml**
- To set params when node starts **ros2 run /package_name /node_name −−ros-args -p argv1:=456 -p argv2:=4567**

ROS1 Nodelets and ROS2 Composition

1. Conceptually node and nodelets (or compositions) are same
2. These are designed to reduce to overhead, i.e., without copying the data, when running on the same machine
3. Quite complicated to implement
4. **Nodelets** (or compositions) are designed to provide a way to run multiple algorithms on a single machine, in a single process, without incurring copy costs when passing messages

ROS Components

Composing multiple nodes in a single process

- To see the available component types **ros2 component types**
- To run the in-build ros2 component manager **ros2 component types**
- To see running component list **ros2 component list**
- To load a component **ros2 component load /ComponentManager ros2_composition composition::SenderNode**
- To unload a component **ros2 component unload /ComponentManager ros2_composition composition::SenderNode**
- **Compile-time composition**, e.g., **ros2 launch ros2_composition composition_demo_launch.py**