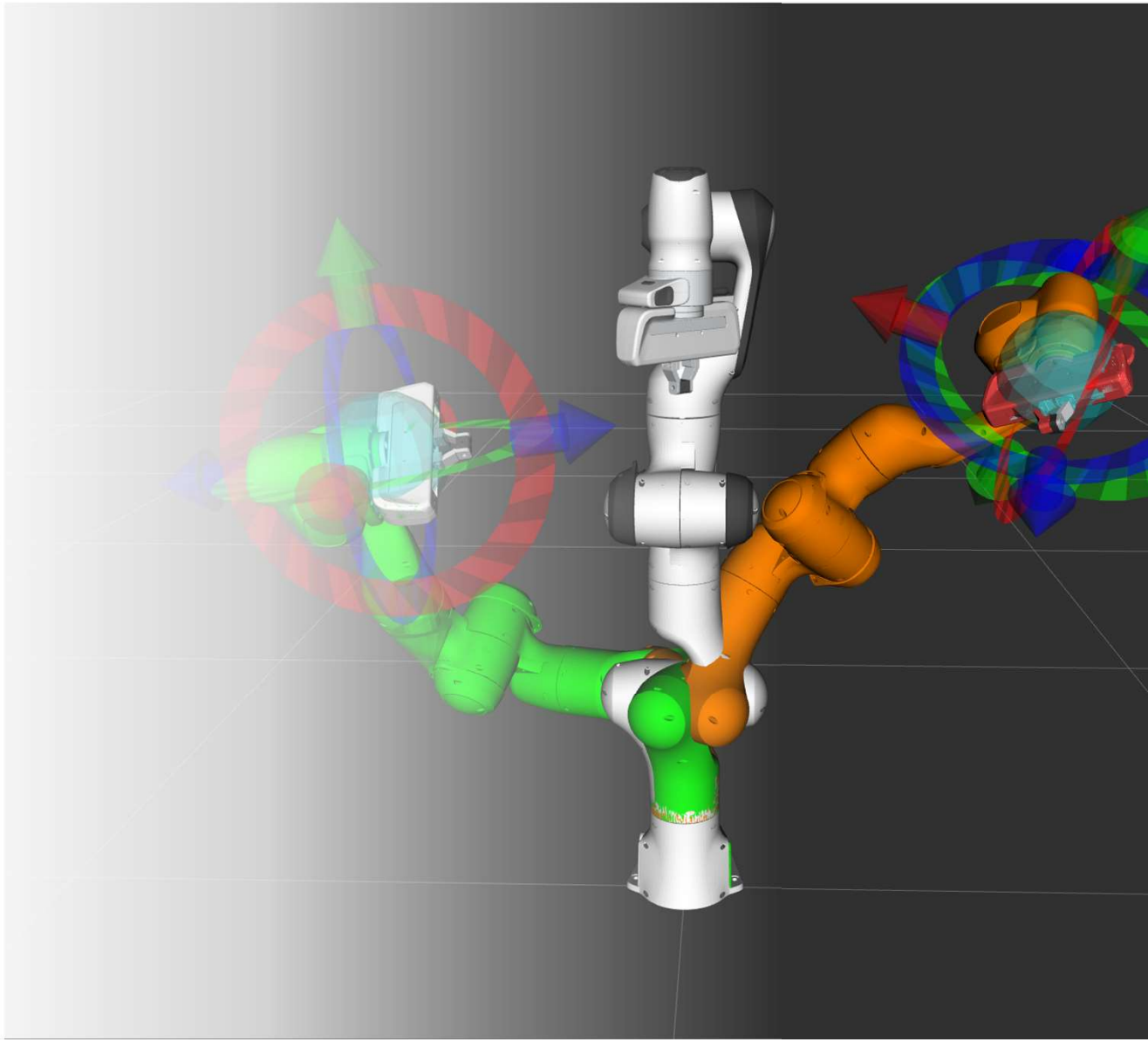
Module 5: ROS Basics



Online Class Etiquette

Be Punctual:

Join the Zoom meeting on time. Just as you would in a physical classroom,

Keep Camera On:

Keep the Camera on at all times, and it's still important to dress as you would for a regular class.

Find a Quiet Space:

Choose a quiet location for attending the session to minimize distractions and background noise.

Check Your Technology:

Test your audio, video, and internet connection before the session begins to avoid technical issues during class time.

Change to real name:

Avoid using nicknames or inappropriate usernames.

Mute Your Microphone:

Keep your microphone muted when you're not speaking to prevent background noise from disrupting the class. Unmute only when you need to ask or answer a question.

Be Respectful:

Treat your instructor and classmates with respect, just as you would in a traditional classroom setting. Avoid making derogatory or offensive comments.

Introduction to ROS

Robotic Operating System

Origin:

ROS originated from Stanford University's Stanford AI Lab (SAIL) in the early 2000s. ROS aimed to simplify the development and collaboration aspects of robotic software to address the lack of a standardized framework in the robotics community. Early contributors, including researchers from Willow Garage, played a crucial role in shaping ROS into what it is today.

Adoption:

ROS gained widespread adoption due to its open-source nature and collaborative community. Researchers, developers, and industries found ROS to be a valuable resource for sharing knowledge and code. Success stories include applications in autonomous vehicles, industrial automation, and research institutions leveraging ROS for cutting-edge projects.

Open Robotics

- Open Robotics is a nonprofit corporation headquartered in Mountain View, California. It is the primary maintainer of the Robot Operating System and the Gazebo simulator. Its stated mission is to support the development, distribution, and adoption of open-source software for use in robotics research, education, and product development
- Notable sources of past and current funding include DARPA, NASA, Amazon, Bosch, Nvidia and the Toyota Research Institute

Brief History

- Willow Garage was gradually dissolved in the ensuing years into several spin-offs, including the Open-Source Robotics Foundation (OSRF), which was created in May 2012 to continue to shepherd the development of ROS and the Gazebo simulator.
- In September 2016, a taxable subsidiary named the Open Source Robotics Corporation (OSRC) was created to foster greater collaboration with industry.
- In December 2022, OSRC and OSRC-SG (the Singapore entity) were acquired by Intrinsic, a subsidiary of Alphabet. OSRF remains an independent non-profit.

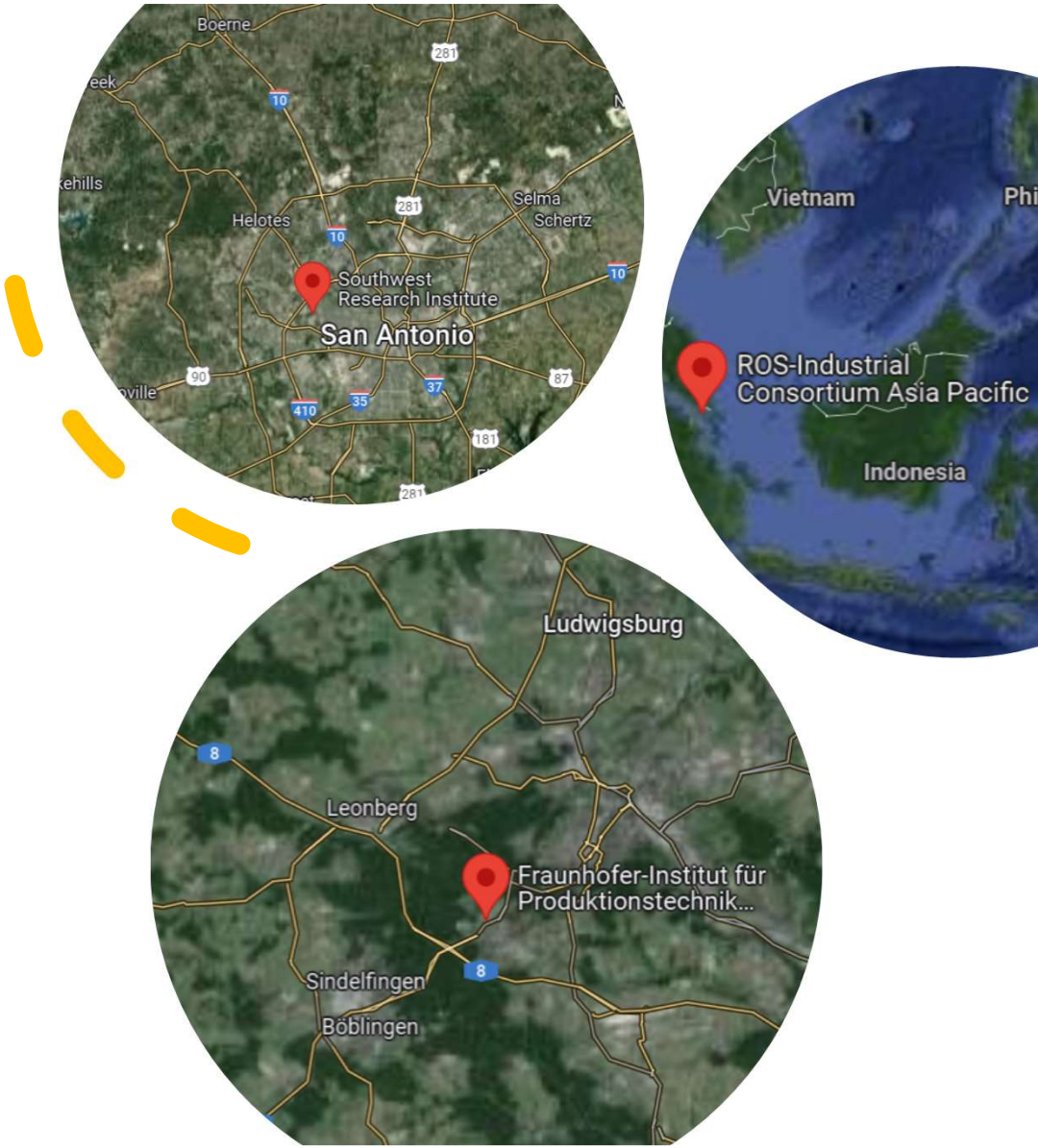
Willow Garage

- Fetch Robotics led by Melonee Wise
- Open Robotics
- Zipline co-founded by Keenan Wyrobek
- Savioke led by Steve Cousin
- Industrial Perception Inc (Acquired by Google in 2013)
- Redwood Robotics (Acquired by Google in 2013)
- Suitable Technologies
- Unbounded Robotics
- ……

ROS-I Consortium

The ROS-Industrial Consortium is a membership organization providing cost-shared applied R&D for advanced factory automation. Consortium members drive new capabilities in ROS-I by championing Focused Technical Projects (FTP) based on their near-term automation requirements.

- ROS-Industrial Asia Pacific Consortium (A*STAR)
- Americas – Consortium (SwRI - Southwest Research Institute)
- Europe – Consortium (Stuttgart, Fraunhofer IPA)



Aspect	Description
Standardization in Robotics	- **Definition:** ROS provides a standardized framework, fostering a common set of tools and conventions in robotics development.
	- **Impact:** This standardization promotes interoperability among diverse robotic systems, encouraging collaboration and knowledge exchange.
	- **Significance:** ROS's role in establishing a shared foundation enhances communication and compatibility, making it easier for developers to work on different robotic projects.
Enhancing Development Efficiency	- **Modularity:** ROS streamlines software development by offering a modular and reusable structure.
	- **Time Efficiency:** Developers can leverage existing ROS packages, reducing the time and effort required for prototyping and building complex robotic applications.
	- **Code Reusability:** The modular nature of ROS components allows developers to reuse code, accelerating the development cycle and minimizing redundancy.

Aspect	Description
Flexibility and Adaptability	- **Versatility:** ROS is flexible and adaptable to various robotic platforms, accommodating different hardware configurations seamlessly.
	- **Compatibility:** ROS's adaptability extends to sensors, actuators, and different robotic architectures, making it a versatile choice for a wide range of robotic applications.
	- **Ease of Integration:** The flexibility of ROS facilitates integration with diverse robotic systems, enabling developers to work with different setups without major modifications.
Contributions to Research	- **Abstraction of Complexity:** ROS abstracts low-level hardware complexities, allowing researchers to focus on high-level algorithmic tasks and experimentation.
	- **Research Advancements:** By simplifying hardware communication, ROS has played a pivotal role in advancing robotics research, enabling innovative projects and breakthroughs.
	- **Facilitating Experimentation:** Researchers benefit from ROS's capabilities, as it provides a conducive environment for experimenting with algorithms and testing novel ideas.

ROS compatibility

- 2d/3d cameras
- laser scanners
- robot actuators
- inertial units
- audio
- GPS
- joysticks
- Arm
- AGV
- Manipulators
- etc.

ROS compatible

ROS1 and ROS2

ROS1 has been around since 2008

- Uses custom TCP/IP middleware

ROS2 is a ground-up reimagining of ROS Started in 2014

- Built on DDS, middleware proven in the industry
- Now on the 9th named release (Iron)

The community is currently in transition!

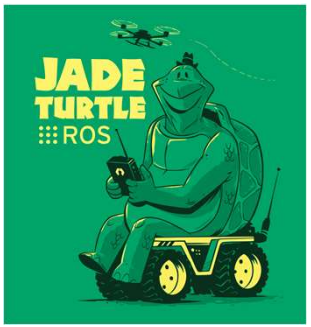
- Final ROS1 release (Noetic) is out (EOL in 2025)
- All critical features are now supported in ROS2

ROS-Industrial will take time to transition

- Many breaking changes / conceptual differences
- Vision is that industrial robots will become native ROS devices

ROS1 Distributions (2014-2019)

2014 ROS Indigo Igloo

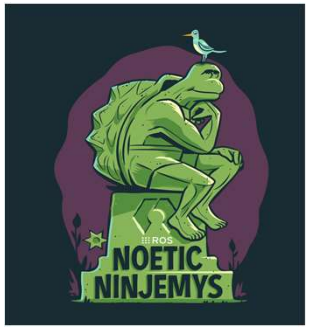


2015 ROS Jade Turtle

2016 ROS Kinetic Kame

2017 ROS Lunar Loggerhead

2018 ROS Melodic Morenia



2019 ROS Noetic Ninjemys

ROS2 Distributions (2017-2023)

2017 ROS2 Ardent Apalone

2018 ROS2 Bouncy Bolson

2018 ROS2 Crystal Clemmys

2019 ROS2 Dashing Diademata

2020 ROS2 Foxy Fitzroy

2021 ROS2 Galactic Geochelone

2022 ROS2 Humble Hawksbill

2023 ROS2 Iron Irwini

ROS1 vs ROS2 - Communication Middleware

ROS1 primarily uses subscriber/publisher communications, which is based on message queuing architecture. It is based on TCP/ IP or User Datagram Protocol (UDP) communication protocol

ROS2, on the other hand, fully embraces DDS as its communication middleware. DDS is an industry-standard middleware that provides a robust and standardized way for distributed systems to communicate.

ROS1 vs ROS2 - Real-time Capabilities

ROS1 lacks native support for real-time systems, which limits its use in applications where real-time performance is critical.

ROS2 has improved support for real-time systems. By using DDS, which is designed to support real-time communication, ROS2 can be more suitable for applications with strict timing requirements.

ROS1 vs ROS2 - Security

ROS2 incorporates security features at its core. With DDS providing a secure communication layer, ROS2 is better equipped to handle security concerns in robotic systems.

ROS1 does not have the same level of built-in security features, making it potentially less suitable for applications with stringent security requirements.

ROS1 vs ROS2 - Multi-robot Systems

ROS2 is designed to better support multi-robot systems. ROS2 provides enhancements to facilitate coordination and communication between robots more efficiently.

While ROS1 can be used in multi-robot scenarios, multi-robot tf over wifi, or tf over large-time-delay teleoperation won't behave optimally out of the box

ROS1 vs ROS2 - Lifecycle Management

ROS2 introduces a more formalized lifecycle management system for nodes. This allows for better control over the initialization, execution, and shutdown of nodes.

ROS1 does not have a standardized lifecycle management system, and developers need to handle node lifecycle manually.

ROS1 vs ROS2 - Language Support

ROS1 primarily supports C++ and has limited support for Python. The ROS1 middleware is tightly coupled with the C++ implementation. Using catkin and calling catkin_python_setup() in a CMakeLists.txt causes a setup.py file to be called

ROS2 has improved support for multiple programming languages, including C++, Python, and potentially others in the future. The middleware abstraction in ROS2 makes it easier to develop and support multiple language bindings.

ROS1 vs ROS2 Community and Development

As of January 2022, ROS2 is actively developed and maintained by the Open Robotics community. ROS1 is still widely used, but the focus of development efforts has shifted to ROS2 for new features and improvements.

While ROS2 builds upon the concepts of ROS1, these differences make ROS2 a more modern and flexible framework, especially suited for evolving robotic applications and addressing some of the limitations of ROS1.

ROS Resources

- ROS-I wiki/ GitHub

https://github.com/ros-industrial

- ROS wiki

https://wiki.ros.org/Documentation

- ROS answers

https://answers.ros.org/questions/

- ROS websites

https://www.ros.org/

- Package wiki, GitHub
- ROSCon, ROS events,

ROS Installation- Debian package

- Setup your computer to accept software from packages.ros.org.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

- Set up your key

```
$ sudo apt install curl # if you haven't already installed curl
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -
```

ROS Installation- Debian package

- make sure your Debian package index is up-to-date

$ sudo apt update

- Desktop-Full Install: (Recommended) : Everything in Desktop plus 2D/3D simulators and 2D/3D perception packages

$ sudo apt install ros-noetic-desktop-full

$ sudo apt install ros-noetic-desktop (**ROS-Base** plus tools like rqt and rviz)

$ sudo apt install ros-noetic-ros-base (ROS packaging, build, and communication libraries. No GUI tools.)

ROS Installation- Debian package

- install a specific package

$ sudo apt install ros-noetic-PACKAGE

Example: $ sudo apt install ros-noetic-universal-robot

- Source this script in every bash terminal you use ROS in.

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

echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc

source ~/.bashrc

ROS Installation- Debian package

- To install other dependencies for building ROS packages

```
$ sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential
```

- Install rosdep that enables you to easily install system dependencies for source

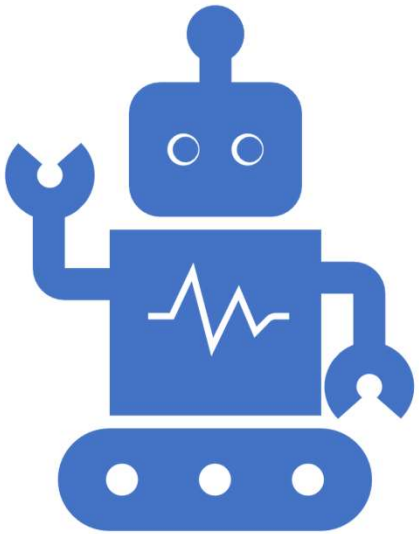
```
$ sudo apt install python3-rosdep
```

```
sudo rosdep init
```

```
rosdep update
```

ROS Installation- Build from source

catkin build and catkin_make are both build tools for managing Catkin workspaces in ROS (Robot Operating System). They serve similar purposes but have some differences in terms of features and usage.



catkin_make

- catkin_make is the older and more established build tool in ROS.
- It is based on the traditional Makefile build system.
- Provides a simple and straightforward way to build packages and entire workspaces.
- It is less flexible than catkin build in terms of parallel builds and advanced features.
- Usage:

```
$ catkin_make
```

catkin build

- catkin build is a newer build tool introduced to address some limitations of catkin_make.
- It is based on the Python catkin_pkg package and has a more modular and extensible design.
- Offers improved parallelization for faster builds and better support for non-CMake projects.
- Provides better integration with ROS tools and is more extensible.
- Usage:

```
$ catkin build
```

Key Differences

Parallelization:

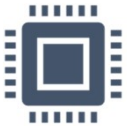
catkin build supports parallel builds by default, enabling faster compilation.

catkin_make can also be configured for parallel builds but requires additional options (**-j** flag).

Extensibility:

catkin build has a more modular and extensible design, making it easier to add plugins and customize the build process.

catkin_make is more monolithic and less extensible.



Workspace Isolation:

catkin build has better workspace isolation, allowing for more reliable and reproducible builds.

catkin_make might have some challenges in maintaining isolation between workspaces.

Tool Integration:

catkin build integrates better with ROS tools, providing more seamless interaction with other ROS commands.

catkin_make is more standalone and may require additional steps for certain ROS functionalities.

Usage Considerations

If you are working with a newer ROS distribution or require advanced features and extensibility, consider using catkin build.

If you are working with an older ROS distribution or have existing projects that use catkin_make, it may be more practical to continue using it for compatibility reasons.

Other tools such as colcon, catkin_make_isolated have similar features so not go into details here.

ROS Installation-Build from source

- Install bootstrap dependencies (Ubuntu)

```
$ sudo apt-get install python3-rosdep python3-rosinstall-generator python3-vcstools python3-vcstool build-essential
```

- Initializing rosdep

```
$ sudo rosdep init
$ rosdep update
```

ROS Installation-Build from source

- Create a catkin Workspace

```
$ mkdir ~/ros_catkin_ws
$ cd ~/ros_catkin_ws
```

- Download the source code for ROS packages (vcstool)

```
$ rosinstall_generator desktop --rosdistro noetic --deps --tar > noetic-desktop.rosinstall
$ mkdir ./src
$ vcs import --input noetic-desktop.rosinstall ./src
```

ROS Installation- Build from source

- Resolving Dependencies

```
$ rosdep install --from-paths ./src --ignore-packages-from-source --rosdistro noetic –y
```

- Building the catkin Workspace

```
$ ./src/catkin/bin/catkin_make_isolated --install -DCMAKE_BUILD_TYPE=Release
```

ROS Installation- Build from source

- Source this setup.bash before running commands

```
$ source ~/ros_catkin_ws/install_isolated/setup.bash
```

Package installation - Debian Installation

- Updates the package index

```
$ sudo apt update
```

- upgrades the actual packages

```
$ sudo apt upgrade
```

- Install the Dependent ROS Packages

```
$ sudo apt-get install ros-melodic-joy ros-melodic-teleop-twist-joy \
  ros-melodic-teleop-twist-keyboard ros-melodic-laser-proc \
  ros-melodic-rgbd-launch ros-melodic-depthimage-to-laserscan \
  ros-melodic-rosserial-arduino ros-melodic-rosserial-python \
  ros-melodic-rosserial-server ros-melodic-rosserial-client \
  ros-melodic-rosserial-msgs ros-melodic-amcl ros-melodic-map-server \
  ros-melodic-move-base ros-melodic-urdf ros-melodic-xacro \
  ros-melodic-compressed-image-transport ros-melodic-rqt* \
  ros-melodic-gmapping ros-melodic-navigation ros-melodic-interactive-markers
```

Package installation - Debian Installation

- Install the ROS package (Turtlebot3 on Melodic)

```
$ sudo apt-get install ros-melodic-dynamixel-sdk
$ sudo apt-get install ros-melodic-turtlebot3-msgs
$ sudo apt-get install ros-melodic-turtlebot3
```

- Set turtlebot model as burger

```
echo "export TURTLEBOT3_MODEL=burger" >> ~/.bashrc
```

- Source file

```
source ~/.bashrc
```

Core Components of ROS

ROS Master: Centralized coordination for ROS nodes, managing topics, services, and parameters.

Nodes: Modular units of computation that perform specific tasks.

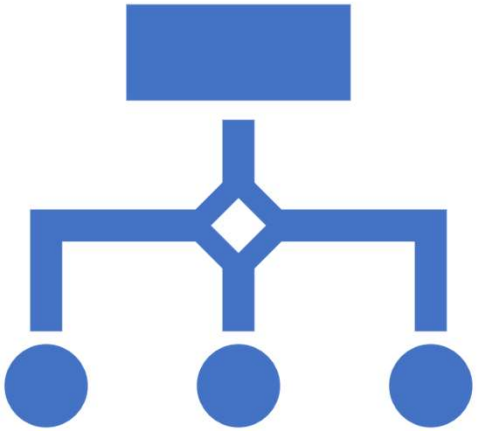
Topics: Communication channels allowing nodes to publish and subscribe to messages.

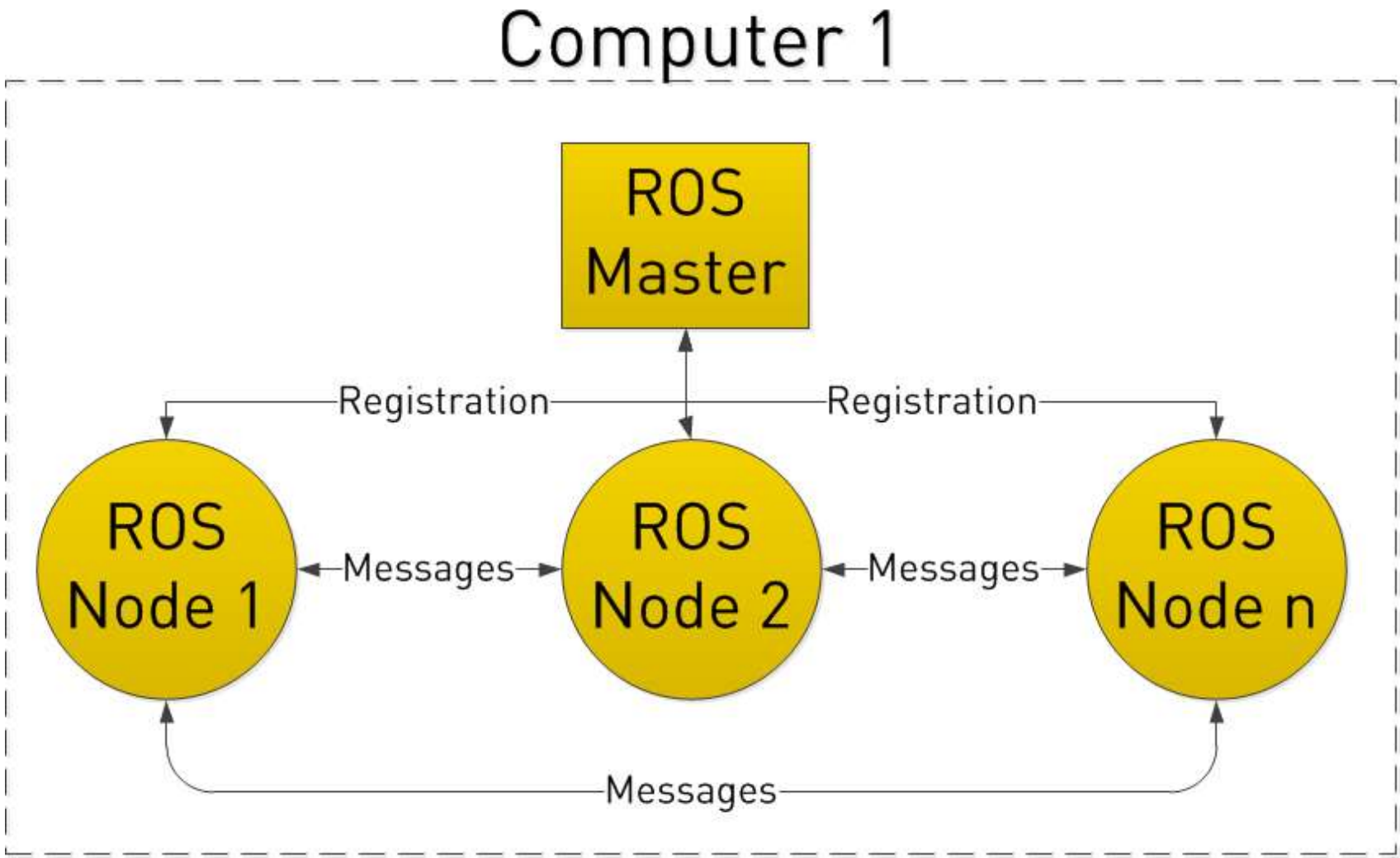
Services: Request-response communication mechanism between nodes.

Actions: Goal-oriented communication for more complex, asynchronous tasks.

ROS Master

- The ROS Master provides naming and registration services for nodes in the ROS system. It allows nodes to discover each other and facilitates communication between them.
- It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.
- The Master is most commonly run using the roscore command, which loads the ROS Master along with other essential components.

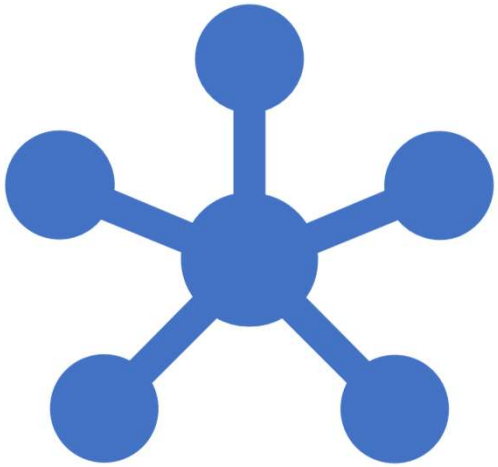



Computer 1
ROS Master
Registration
Registration
ROS Node 1
Messages
ROS Node 2
Messages
ROS Node n
Messages

ROS Architecture: Nodes

- Nodes are the basic computational units in ROS. A ROS system is comprised of multiple nodes, each representing a distinct process. Nodes communicate with each other by passing messages.

- A Node is a standalone piece of functionality
  - Most communication happens between nodes
  - Nodes can run on many different devices
  - Often one node per process, but not always



Topics

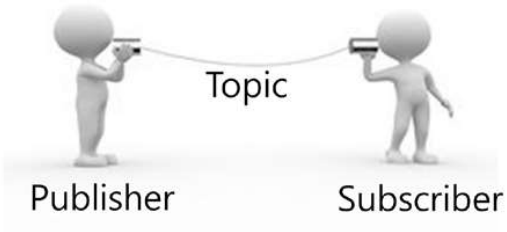
Topics are named buses over which nodes exchange messages. Nodes can publish messages to a topic or subscribe to receive messages from a topic. Topics enable the decoupling of nodes and facilitate communication.

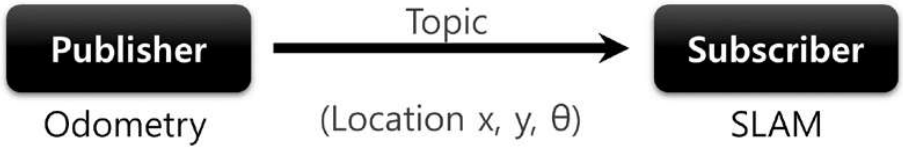
List all the topics

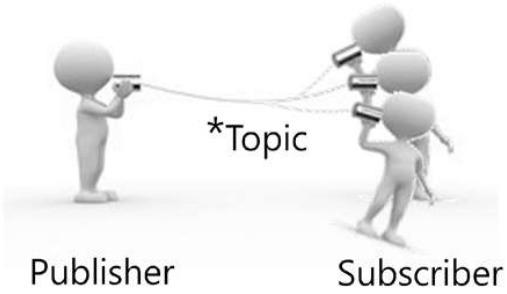
```
$ rostopic list
```

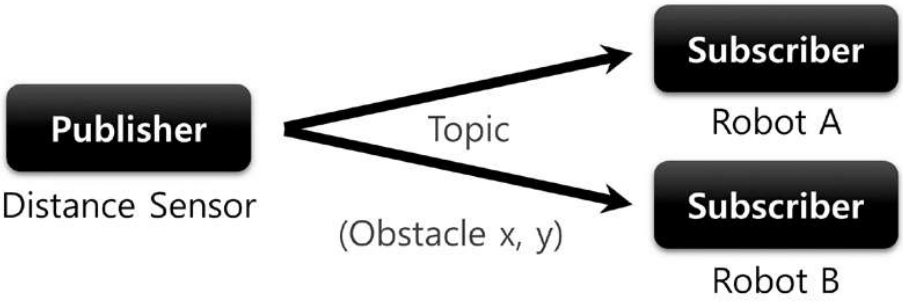
display Messages published to /topic_name

```
$ rostopic echo /topic_name
```





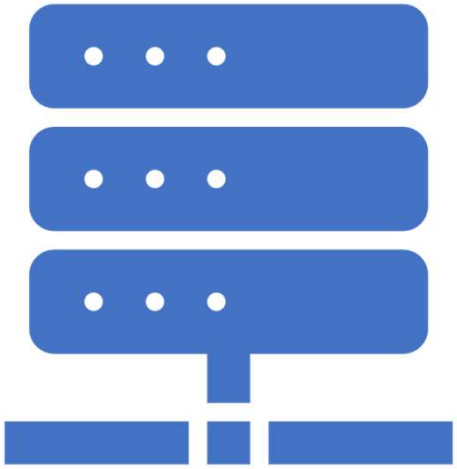




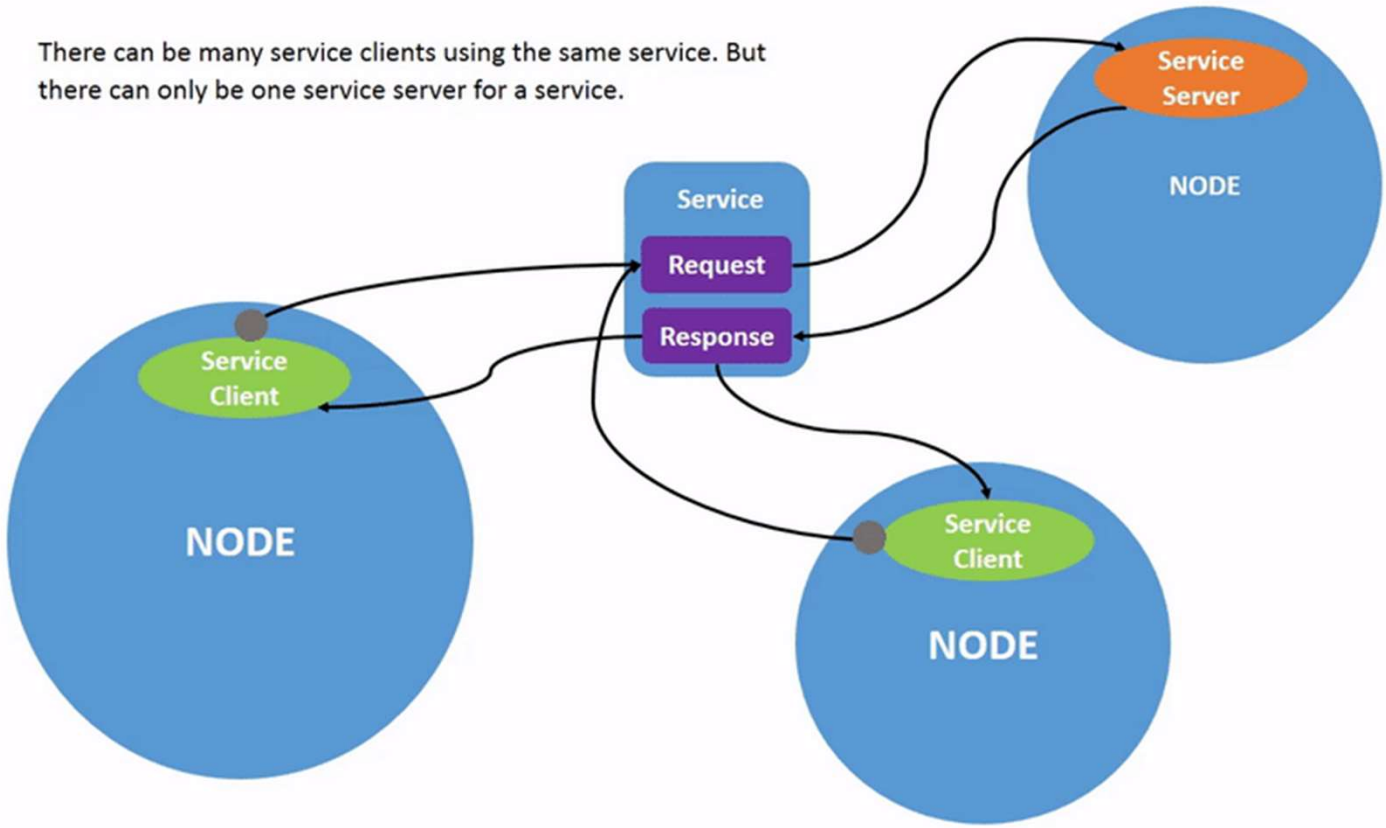
*Topic not only allows 1:1 Publisher and Subscriber communication, but also supports 1:N, N:1 and N:N depending on the purpose.

Services

- Services allow nodes to call functions in other nodes. A service has a request and a response, and it provides a synchronous communication pattern between nodes.
- Request / reply is done via a Service, which is defined by a pair of messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply.
- Services are defined using srv files, which are compiled into source code by a ROS client library.



There can be many service clients using the same service. But there can only be one service server for a service.

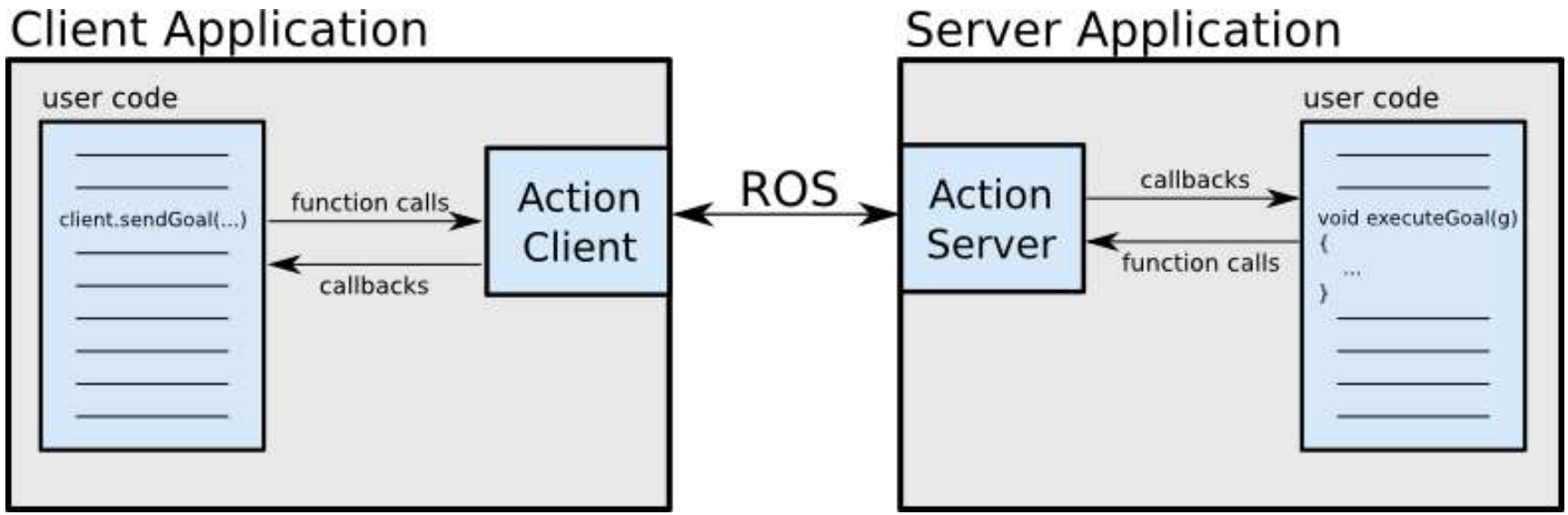


Actions

The actionlib package provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

The ActionClient and ActionServer communicate via a "ROS Action Protocol", which is built on top of ROS messages. The client and server then provide a simple API for users to request goals (on the client side) or to execute goals (on the server side) via function calls and callbacks.

Action Specification: Goal + Feedback + Result


Client Application
user code
client.sendGoal(...)
function calls
callbacks
Action Client
ROS
Action Server
Server Application
user code
callbacks
function calls
void executeGoal(g)
{
...
}

Tools and Utilities

roscpp and rospy: ROS client libraries for C++ and Python.

roslaunch: Tool for launching multiple ROS nodes with a single command.

RViz: Visualization tool for debugging and monitoring robotic systems.

RQT: A modular GUI framework for robot development.

roscpp and rospy

roscpp and rospy

- roscpp and rospy are two libraries in ROS (Robot Operating System) that provide support for programming in C++ and Python, respectively. They are used for creating ROS nodes, interacting with the ROS middleware, and facilitating communication between nodes.

roscpp (ROS C++ Library)

roscpp is the ROS library for C++ programming.

It allows developers to create and manage ROS nodes, publish and subscribe to topics, provide and call services, and manage the ROS parameter server in C++.

Key components in roscpp include NodeHandle for creating and managing nodes, Publisher and Subscriber for communication, and ServiceServer and ServiceClient for providing and calling services.

Example (C++)

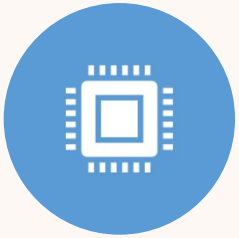
```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle nh;

  ros::Publisher chatter_pub = nh.advertise<std_msgs::String>("chatter", 10);

  ros::Rate loop_rate(10);

  while (ros::ok())
  {
    std_msgs::String msg;
    msg.data = "Hello, ROS!";
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }

  return 0;
}
```

rospy (ROS Python Library)

rospy is the ROS library for Python programming.

It provides similar functionality to roscpp but is tailored for Python developers.

Developers can create ROS nodes, publish and subscribe to topics, provide and call services, and manage the ROS parameter server using rospy.



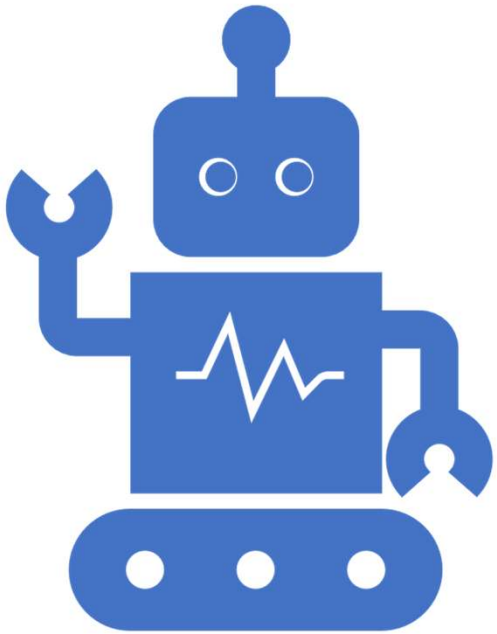
rospy uses Python concepts such as callbacks and Python data types.

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

    while not rospy.is_shutdown():
        hello_str = "Hello, ROS!"
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Example (Python)

Conclusion

Both roscpp and rospy enable developers to create robust and modular robotic systems by providing a convenient interface for ROS concepts such as nodes, topics, services, and parameters. The choice between roscpp and rospy depends on the developer's language preference (C++ or Python) and the specific requirements of the project. It's common to have a combination of both languages within a larger ROS system, with nodes communicating seamlessly across the two languages.



roslaunch

roslaunch

roslaunch is a command-line tool in ROS that is used to launch and manage ROS nodes, launch files, and entire ROS systems. It provides a convenient way to start multiple nodes with specific parameters, remappings, and configurations in a structured and repeatable manner. roslaunch is particularly useful for managing complex robotic systems composed of multiple nodes.

roslaunch

Launch Files:

- roslaunch uses XML-based launch files to describe the configuration of a set of nodes.
- These launch files contain information about which nodes to launch, their parameters, remappings, and other configurations.

Syntax:

- The basic syntax of a roslaunch command is:

```
$ roslaunch package_name launch_file.launch
```

package_name is the name of the ROS package, launch_file.launch is launch file within that package

roslaunch

Parameterization:

- Launch files support parameterization, allowing you to pass arguments and parameters to nodes during launch.
- Parameters can be set in the launch file or dynamically loaded from ROS parameter server.

Remapping:

- roslaunch enables remapping of node names and topics, allowing for easy configuration and reuse of launch files.

Node Grouping:

- Nodes can be grouped into different namespaces, allowing for better organization and avoiding naming conflicts.

roslaunch

Conditional Launching:

- Conditional execution of nodes or entire sections of a launch file based on conditions.

rosrun

rosrun

rosrun is a command-line tool in ROS (Robot Operating System) used to run individual ROS nodes. It allows you to run a specific node by specifying its package name and node name.

rosrun

Syntax:

- The basic syntax of a roslaunch command is:

```
$ rosrun [package_name] [node_name]
```

package_name is the name of the ROS package, node_name is specific node that you would like to run separately

Example:

```
$ rosrun my_robot_package robot_controller
```

Example

```
<launch>
        <!-- specify the planning pipeline -->
        <arg name="pipeline" default="ompl" />
        <!-- By default, we do not start a database (it can be large) -->
        <arg name="db" default="false" />
        <!-- Allow user to specify database location -->
        <arg name="db_path" default="$(find ur10e_moveit_config)/default_warehouse_mongo_db" />
        <!-- By default, we are not in debug mode -->
        <arg name="debug" default="false" />
        <!-- By default, we will load or override the robot_description -->
        <arg name="load_robot_description" default="true"/>
        <arg name="use_gui" default="false" />
        <arg name="use_rviz" default="true" />
        <!-- If needed, broadcast static tf for robot root -->
        <!-- We do not have a robot connected, so publish fake joint states -->
        <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" u
                <rosparam param="source_list">[move_group/fake_controller_joint_states]</rosparam>
        </node>
        <node name="joint_state_publisher" pkg="joint_state_publisher_gui" type="joint_state_publishe
                <rosparam param="source_list">[move_group/fake_controller_joint_states]</rosparam>
        </node>
        <!-- Given the published joint states, publish tf for the robot links -->
        <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" r
        <!-- Run the main MoveIt! executable without trajectory execution (we do not have controllers
        <include file="$(find ur10e_moveit_config)/launch/move_group.launch">
                <arg name="allow_trajectory_execution" value="true"/>
                <arg name="fake_execution" value="true"/>
                <arg name="info" value="true"/>
                <arg name="debug" value="$(arg debug)"/>
                <arg name="pipeline" value="$(arg pipeline)"/>
                <arg name="load_robot_description" value="$(arg load_robot_description)"/>
        </include>
        <!-- Run Rviz and load the default config to see the state of the move_group node -->
        <include file="$(find ur10e_moveit_config)/launch/moveit_rviz.launch" if="$(arg use_rviz)">
                <arg name="config" value="$(find ur10e_moveit_config)/launch/moveit.rviz"/>
                <arg name="debug" value="$(arg debug)"/>
        </include>
        <!-- If database loading was enabled, start mongodb as well -->
        <include file="$(find ur10e_moveit_config)/launch/default_warehouse_db.launch" if="$(arg db)'
                <arg name="moveit_warehouse_database_path" value="$(arg db_path)"/>
        </include>
</launch>
```

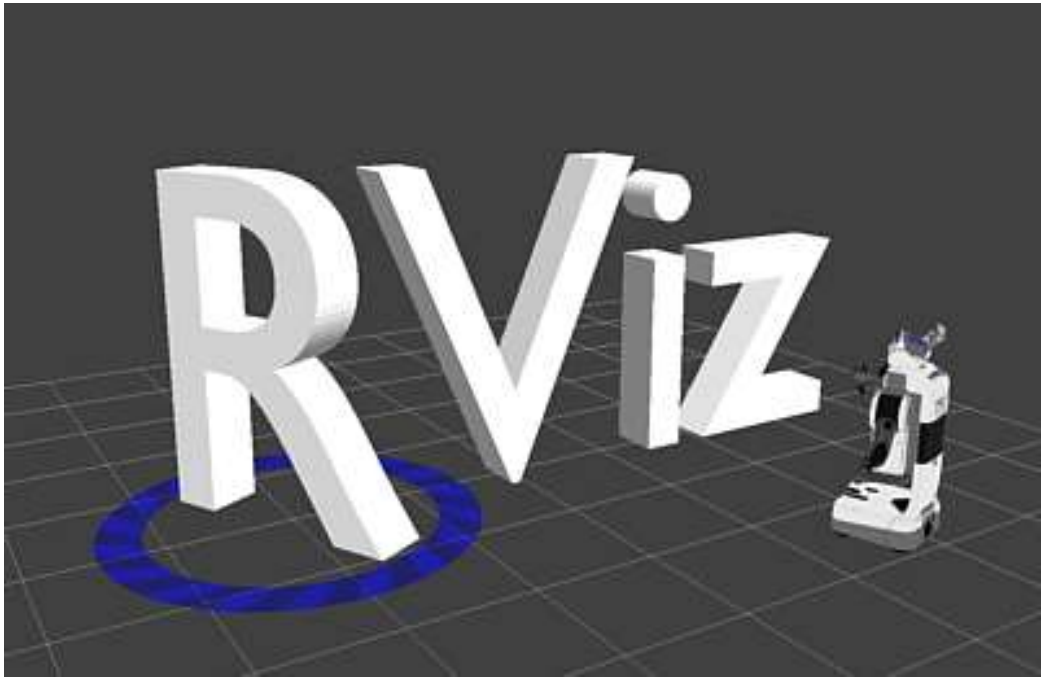
Roslaunch Benefits:

Modularity: roslaunch promotes modularity by allowing the configuration of each node to be encapsulated in separate launch files.

Reproducibility: Launch files make it easy to reproduce a specific configuration of a robotic system.

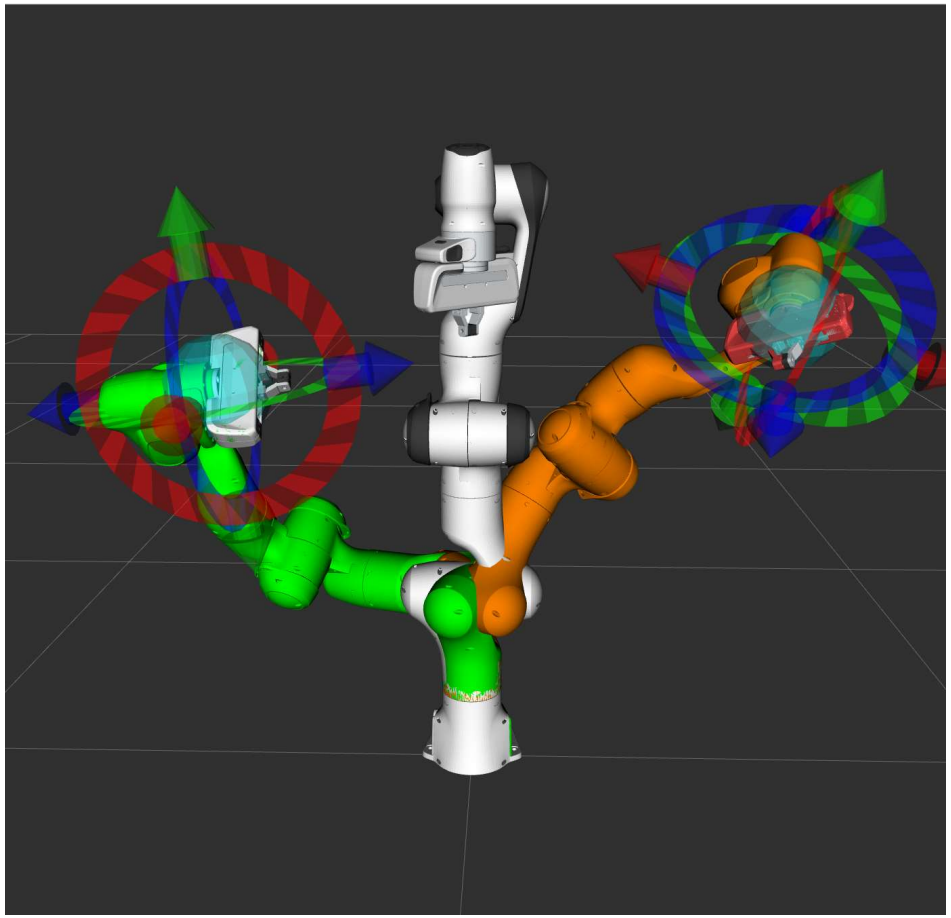
Convenience: roslaunch simplifies the process of starting multiple nodes and managing their configurations, making it a powerful tool for ROS developers.

RViz



RViz

RViz, short for ROS Visualization, is a powerful 3D visualization tool in ROS. It provides a real-time graphical interface for visualizing sensor data, robot models, and other information from a ROS system. RViz is a crucial tool for debugging, monitoring, and understanding the state of a robotic system.



Rviz Key Features

3D Visualization:	RViz allows users to visualize 3D representations of the robot model, sensor data, and the environment.
Robot Models:	Users can load and display robot models to visualize the physical structure of the robot.
Sensor Data Display:	RViz supports the visualization of various sensor data, including laser scans, point clouds, camera images, and more.
TF (Transform) Visualization:	RViz can display coordinate frame transforms using the TF (Transform) library, helping users understand the relationship between different parts of the robot.

Rviz Key Features

Interactive Markers	Interactive markers allow users to manipulate the robot's pose, plan paths, and perform other interactive tasks directly in RViz.
Configuration Panels	RViz provides configuration panels that allow users to customize the visualization settings, including robot models, sensor displays, and TF frames
ROS Topics Integration	RViz subscribes to ROS topics to receive real-time data, making it easy to integrate with different ROS nodes.
Tools for Analysis	RViz includes tools for measuring distances, angles, and other properties in the 3D visualization.
Plugins	RViz is extensible through the use of plugins, allowing users to add custom displays and functionalities.

Rviz Common Use Cases

Robot State Visualization:

- Check the current state of the robot, including joint positions, velocities, and other state information.

Sensor Data Inspection:

- Visualize sensor data such as laser scans, point clouds, and camera images to verify their correctness and accuracy.

Path Planning Visualization:

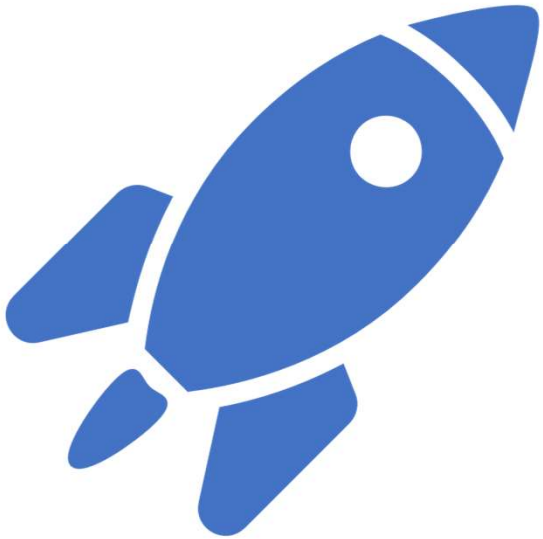
- Evaluate and visualize the output of path planning algorithms by displaying planned paths and trajectories.

Debugging and Analysis:

- Use RViz as a debugging tool to identify issues in the robot's perception, localization, and control.

Simulation Analysis:

- Integrate RViz with robot simulations to visualize the simulated robot's behavior and environment.



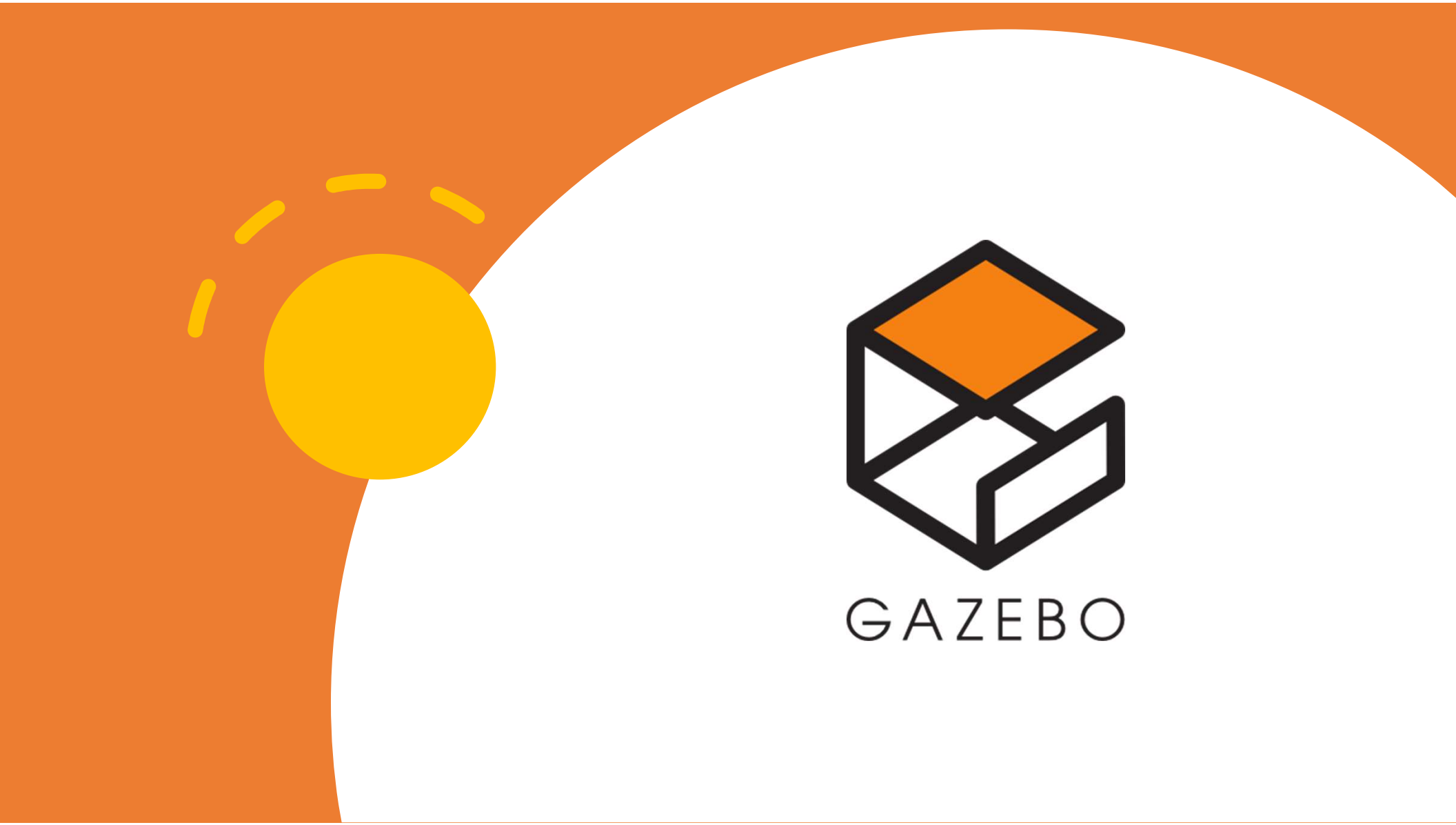
Usage

- To launch RViz, you can use the following command

```
$ rosrun rviz rviz
```

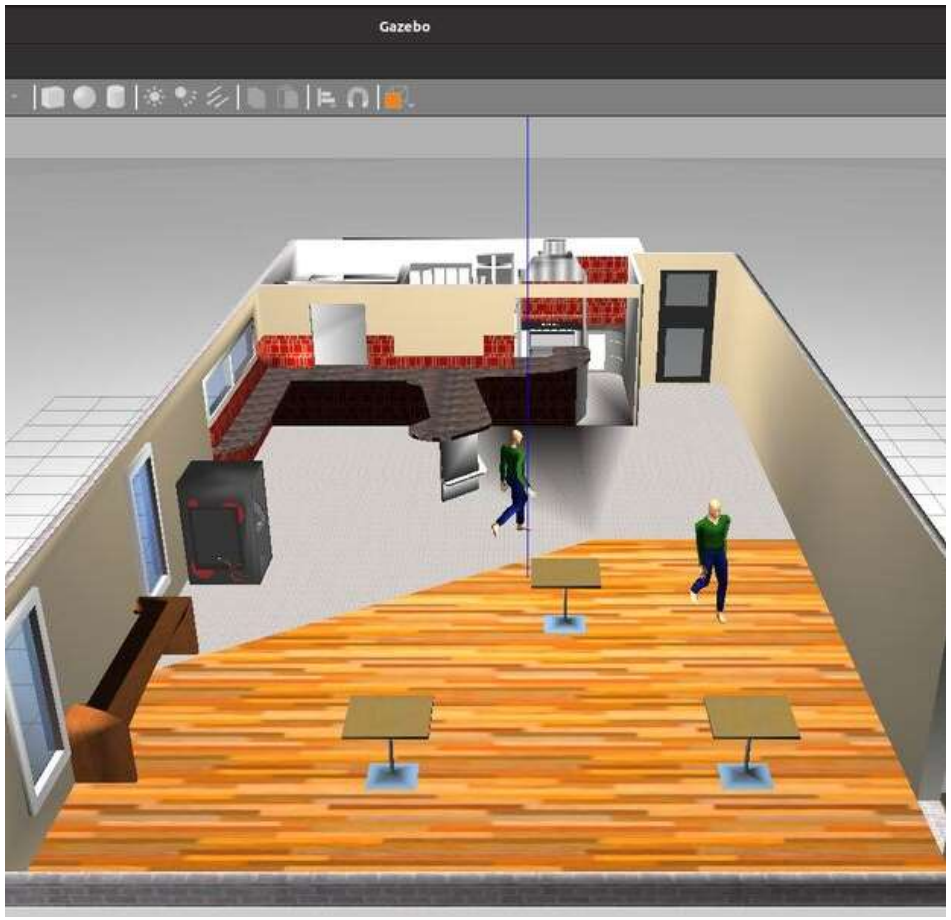
- Launch with a specific configuration file

```
$ rosrun rviz rviz -d your_config_file.rviz
```


GAZEBO

Gazebo

Definition: Gazebo is an open-source 3D simulation environment widely used in the robotics community for simulating robots, environments, and various scenarios. It provides a platform for testing and validating robot designs, control algorithms, and sensor integration in a realistic and dynamic simulated environment.



Key features

Physics Simulation:

Gazebo incorporates a robust physics engine that simulates realistic interactions between objects, including rigid body dynamics, collision detection, and friction.

Sensor Simulation:

It supports the simulation of various sensors commonly used in robotics, such as cameras, LIDAR, GPS, IMU, and more. This allows developers to test and validate sensor configurations in a virtual environment.

Robot Models:

Gazebo enables users to import and simulate different robot models, supporting a variety of robot types including wheeled robots, legged robots, and robotic arms. Robot models can be customized and integrated into the simulation.

Scenarios and Environments:

Users can create custom environments and scenarios for testing specific robot behaviors. Gazebo includes a library of pre-built models, terrains, and objects that can be used or customized for simulations.

ROS Integration:

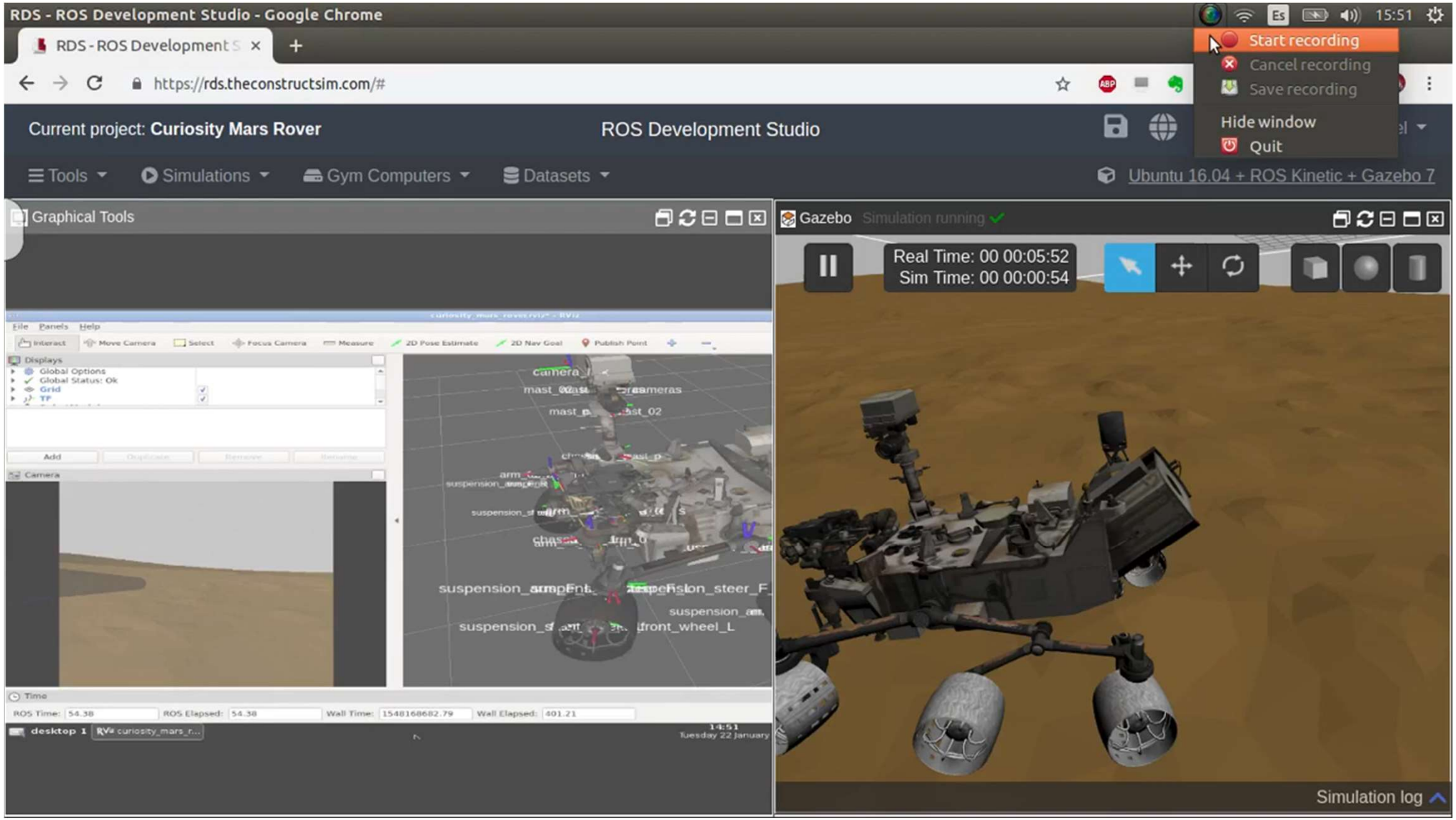
Gazebo is commonly used in conjunction with ROS (Robot Operating System). It provides a bridge to connect simulated robots and sensors with ROS nodes, allowing seamless integration into ROS-based robotics workflows.

Applications

Robotics Research and Development:	• Gazebo is extensively used for prototyping, testing algorithms, and validating control strategies in a simulated environment before deploying them to physical robots.
Education:	• It serves as an educational tool for teaching robotics concepts and principles. Students can experiment with different robot designs and scenarios without the need for physical hardware.
Autonomous Systems Testing:	• Developers use Gazebo to simulate and evaluate the performance of autonomous systems, including self-driving cars and drones, in complex and dynamic environments.
Multi-Robot Simulations:	• Gazebo supports the simulation of multiple robots interacting with each other, making it suitable for scenarios involving swarm robotics or collaborative tasks.

Rviz vs Gazebo

Aspect	RViz	Gazebo
Type	**Visualization Tool:** RViz is primarily a visualization tool used for visualizing and analyzing sensor data, robot states, and other information from a robotic system.	**Simulation Environment:** Gazebo is a 3D simulation environment used for simulating robots, environments, and scenarios. It provides a platform for testing and validating robot designs and control algorithms in a simulated environment.
Purpose	**Data Visualization:** RViz is designed for visualizing information gathered from sensors and the internal state of a robot. It is often used for debugging, monitoring, and gaining insights into the robot's perception and planning.	**Robot Simulation:** Gazebo is designed for simulating the physical aspects of robots and their interactions with the environment. It is used for testing and validating robot behaviors in a realistic and dynamic simulation.
Features	•**Sensor Displays:** RViz supports the visualization of data from various sensors, such as cameras, LIDAR, and depth sensors. •**Robot Model Visualization:** It allows the display of a robot's 3D model, providing a visual representation of its configuration. •**Planned Trajectories:** RViz can visualize planned trajectories, helping developers and researchers assess the robot's planned path.	•**Physics Simulation:** Gazebo includes a robust physics engine that simulates realistic interactions between objects, providing accurate representations of robot dynamics. •**Sensor Simulation:** It supports the simulation of various sensors, allowing developers to test sensor configurations and evaluate the performance of sensor-based algorithms.
Integration	**ROS Integration:** RViz is an integral part of the Robot Operating System (ROS) ecosystem and is commonly used in conjunction with other ROS nodes for data visualization.	**ROS Integration:** Gazebo is commonly used with ROS, providing a bridge for simulating robots and sensors in a ROS-based environment.
Applications	**Debugging and Analysis:** RViz is particularly useful for debugging sensor data, analyzing robot state, and validating the performance of perception and planning algorithms.	•**Robotics Research and Development:** Gazebo is extensively used for prototyping, testing algorithms, and validating control strategies in a simulated environment before deploying them to physical robots. •**Autonomous Systems Testing:** Developers use Gazebo to simulate and evaluate the performance of autonomous systems, including self-driving cars and drones, in complex and dynamic environments.


RDS - ROS Development Studio - Google Chrome
RDS - ROS Development S
https://rds.theconstructsim.com/#
15:51
Start recording
Cancel recording
Save recording
Hide window
Quit
Current project: Curiosity Mars Rover
ROS Development Studio
Tools
Simulations
Gym Computers
Datasets
Ubuntu 16.04 + ROS Kinetic + Gazebo 7
Graphical Tools
Gazebo
Simulation running
Real Time: 00 00:05:52
Sim Time: 00 00:00:54
Simulation log

rqt

rqt - ROS Qt-based GUI Framework

rqt is a Qt-based graphical user interface (GUI) framework in ROS. It provides a modular and extensible architecture for developing graphical tools that can be used for various ROS-related tasks. rqt tools are commonly used for monitoring, debugging, and visualizing information from a running ROS system.

rqt - Key Features

Plugin Architecture:	• **rqt** follows a plugin-based architecture, allowing developers to create and integrate custom plugins for specific functionalities.
Visualization Tools:	• **rqt** provides a variety of visualization tools for displaying data, including plots, images, and interactive displays.
Logging and Console Tools:	• Tools for monitoring ROS log messages and viewing console outputs of running nodes.
Graphical Node Management:	• **rqt** allows users to manage and monitor the state of ROS nodes graphically.

rqt - Key Features

Integrated Development Environment (IDE):	• Some **rqt** plugins serve as simple IDEs for writing and debugging ROS code.
Parameter Configuration:	• Tools for configuring and monitoring ROS parameters in a graphical interface.
Topic and Service Inspection:	• Inspection tools for visualizing the messages being published on ROS topics and the services available in the system.
Integrated Shell:	• **rqt** includes an integrated shell for running ROS commands and interacting with the system.

Common rqt tools

rqt_graph: Visualizes the ROS computation graph, showing the connections between nodes and topics.

rqt_plot: Provides a tool for plotting real-time data from ROS topics.

rqt_bag: Allows users to inspect and replay data recorded in ROS bag files.

rqt_console: Monitors and displays ROS log messages in a console-like interface.

rqt_image_view: Displays images from ROS image topics.

rqt_reconfigure: Provides an interface for dynamically reconfiguring parameters of ROS nodes.

Usage

- To launch rqt

```
$ rqt
```

- To launch a specific rqt plugin

```
$ rqt -s plugin_name
```

- Load multiple plugins simultaneously

```
$ rqt -s "plugin_name_1,plugin_name_2"
```

Benefits

Modularity: rqt promotes modularity by allowing the integration of different plugins for specific tasks, making it flexible for various use cases.

Extensibility: Developers can create custom plugins to extend the functionality of **rqt** for their specific needs.

Visualization and Debugging: rqt tools provide a graphical interface for visualizing and debugging aspects of the ROS system, which is particularly valuable for complex robotic systems.