ROS2 Humble: Complete Guide

Robot Operating System 2 - From Basics to Advanced

CS498GC Mobile Robotics

University of Illinois

Fall 2025

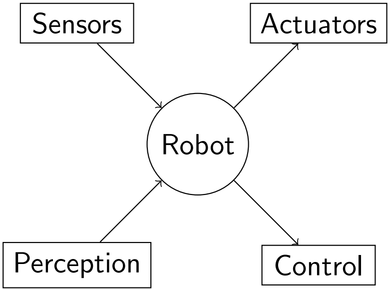
Presented by: Kulbir Singh Ahluwalia
Date: September 10, 2025

Course Overview

1. Introduction to ROS2
2. Installation and Setup
3. ROS2 Nodes
4. Topics and Messages
5. Services
6. Actions
7. Parameters
8. Launch Files
9. TF2 Transform System
10. Navigation Stack (Nav2)
11. Perception and Sensors
12. Simulation with Gazebo
13. Best Practices and Patterns
14. Debugging and Development Tools
15. Practical Exercises
16. Project Ideas
17. Advanced Topics

What is ROS2?

- **Robot Operating System 2**
- Open-source framework for robotics
- Middleware for robot software development
- Distributed computing architecture
- Language-agnostic (C++, Python, etc.)
- Industry-standard for robotics



Why ROS2 Over ROS1?

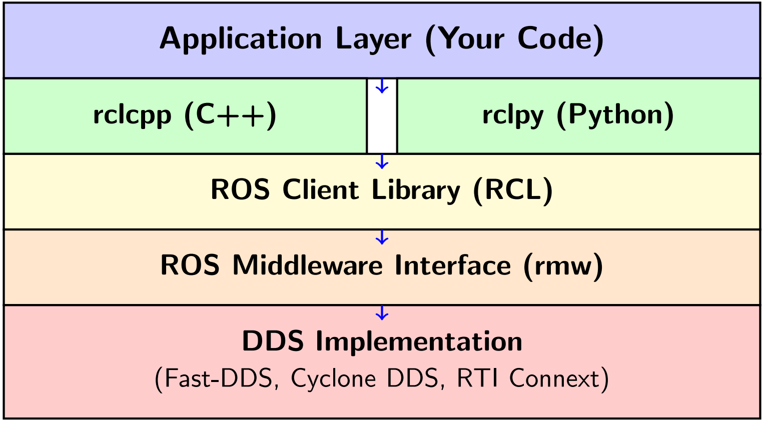
ROS1 Limitations:

- Single point of failure (master)
- No real-time support
- Limited security features
- TCP/IP only communication
- Ubuntu-centric

ROS2 Improvements:

- DDS-based (distributed)
- Real-time capable
- Built-in security
- Multiple transport protocols
- Cross-platform support
- Better performance
- Industry 4.0 ready

ROS2 Architecture



Key Concepts in ROS2

- **Nodes**: Independent processes that perform computation
- **Topics**: Named buses for asynchronous communication
- **Services**: Synchronous request/reply interactions
- **Actions**: Long-running tasks with feedback
- **Parameters**: Configuration values for nodes
- **Launch Files**: System configuration and startup
- **Packages**: Organization units for ROS2 code
- **Workspaces**: Development environments

Installing ROS2 Humble on Ubuntu 22.04

```
# Set locale
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8

# Setup sources
sudo apt install software-properties-common
sudo add-apt-repository universe

# Add ROS2 GPG key
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.
    key -o /usr/share/keyrings/ros-archive-keyring.gpg

# Add repository to sources list
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/
    ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os
    -release && echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.
    list.d/ros2.list > /dev/null
```

Installing ROS2 Packages

```
# Update and install ROS2
sudo apt update
sudo apt upgrade
sudo apt install ros-humble-desktop

# Development tools
sudo apt install ros-dev-tools

# Source the setup script
source /opt/ros/humble/setup.bash

# Add to bashrc for automatic sourcing
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc

# Test installation
ros2 doctor
```

Docker-based ROS2 Setup

```bash
# Install Docker (Ubuntu)
sudo apt update && sudo apt install docker.io
sudo usermod -aG docker $USER

# Pull ROS2 Humble image
docker pull ros:humble-desktop

# Run ROS2 container with GUI support
docker run -it --rm \
  --net=host \
  --env="DISPLAY" \
  --volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
  ros:humble-desktop

# Create persistent workspace container
docker run -it --name ros2_dev \
  -v ~/ros2_ws:/ros2_ws \
  ros:humble-desktop
```

Creating a ROS2 Workspace

```bash
# Create workspace directory
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws

# Build the workspace (even if empty)
colcon build

# Source the workspace
source ~/ros2_ws/install/setup.bash

# Add to bashrc
echo "source ~/ros2_ws/install/setup.bash" >> ~/.bashrc

# Install colcon extensions
sudo apt install python3-colcon-common-extensions
```

Understanding ROS2 Nodes

- Fundamental building blocks of ROS2 systems
- Each node is a single-purpose, modular process
- Nodes communicate via topics, services, and actions
- Can be written in C++ or Python
- Managed by the ROS2 executor

Node Characteristics:

- Has a unique name in the system
- Can have parameters
- Can publish/subscribe to topics
- Can provide/use services
- Can create/use actions

Creating a Simple Node - Python

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node

class MinimalNode(Node):
    def \\_\\_init\\_\\_(self):
        super().\\_\\_init\\_\\_('minimal\\_node')
        self.get\\_logger().info('Node has been started')

        # Create a timer that calls a callback every second
        self.timer = self.create\\_timer(1.0, self.timer\\_callback)
        self.counter = 0

    def timer\\_callback(self):
        self.get\\_logger().info(f'Counter: \\{self.counter\\}')
        self.counter += 1

def main(args=None):
    rclpy.init(args=args)
    node = MinimalNode()
```

Creating a Simple Node - C++

```
#include <rclcpp/rclcpp.hpp>
#include <chrono>

class MinimalNode : public rclcpp::Node \\{
public:
    MinimalNode() : Node("minimal\\_node"), counter\\_(0) \\{
        RCLCPP\\_INFO(this->get\\_logger(), "Node has been started");

        timer\\_ = this->create\\_wall\\_timer(
            std::chrono::seconds(1),
            std::bind(\\&MinimalNode::timer\\_callback, this));
    \\}

private:
    void timer\\_callback() \\{
        RCLCPP\\_INFO(this->get\\_logger(), "Counter: \\%d", counter\\_++);
    \\}

    rclcpp::TimerBase::SharedPtr timer\\_;
    int counter\\_;
```

Node Management Commands

```
# List all running nodes
ros2 node list

# Get info about a specific node
ros2 node info /node\\_name

# Run a node from a package
ros2 run package\\_name node\\_name

# Run with remapped name
ros2 run package\\_name node\\_name --ros-args -r \\_\\_node:=new\\_name

# Run with parameters
ros2 run package\\_name node\\_name --ros-args -p param\\_name:=value
```

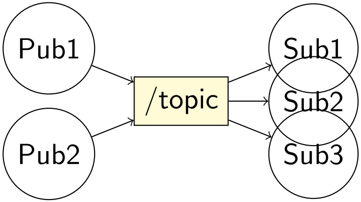
Topics: Publish-Subscribe Pattern

Characteristics:

- Asynchronous communication
- Many-to-many connections
- Typed message passing
- Best for continuous data streams
- Decoupled publishers and subscribers

Use Cases:

- Sensor data streaming
- Robot state broadcasting
- Command velocity
- Image/video streams



Publisher Example - Python

```
import rclpy
from rclpy.node import Node
from std\\_msgs.msg import String

class PublisherNode(Node):
    def __init__(self):
        super().__init__('publisher_node')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        self.timer = self.create_timer(0.5, self.publish_message)
        self.count = 0

    def publish_message(self):
        msg = String()
        msg.data = f'Hello ROS2! Count: \{self.count\}'
        self.publisher_.publish(msg)
        self.get_logger().info(f'Publishing: "\{msg.data\}"')
        self.count += 1
```

Subscriber Example - Python

```python
import rclpy
from rclpy.node import Node
from std\\_msgs.msg import String

class SubscriberNode(Node):
    def __init__(self):
        super().__init__('subscriber_node')
        self.subscription = self.create_subscription(
            String,
            'chatter',
            self.listener_callback,
            10)

    def listener_callback(self, msg):
        self.get_logger().info(f'Received: "\{msg.data\}"')

def main(args=None):
    rclpy.init(args=args)
    node = SubscriberNode()
    rclpy.spin(node)
```

Common ROS2 Message Types

Standard Messages:

```
# std_msgs
Bool, Int32, Float64, String
Header, Time

# geometry_msgs
Point, Pose, Twist
Transform, Quaternion
PoseStamped, TwistStamped

# sensor_msgs
Image, PointCloud2
LaserScan, Imu
JointState, NavSatFix
```

Custom Message Example:

```
# RobotStatus.msg
std_msgs/Header header
string robot_name
float64 battery_level
bool is_active
geometry_msgs/Pose current_pose
float64[] joint_positions
string status_message
```

Topic Commands

```
# List all active topics
ros2 topic list

# List with message types
ros2 topic list -t

# Show topic info
ros2 topic info /topic_name

# Echo messages from a topic
ros2 topic echo /topic_name

# Show message frequency
ros2 topic hz /topic_name

# Publish to a topic from command line
ros2 topic pub /topic\\_name std\\_msgs/msg/String "data: 'Hello'"

# Record topics to a bag file
ros2 bag record /topic1 /topic2
```

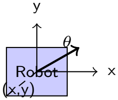
ROS2 Messages: Definition and Structure

What are ROS2 Messages?

- **Data structures** that define how information is formatted and transmitted
- **Strongly typed** - each message has a specific format and field types
- **Platform independent** - same message works across different systems
- **Serializable** - converted to bytes for network transmission
- **Self-describing** - contain metadata about their structure

Real Robot Example: TurtleBot Pose

- **Message Type**: `geometry_msgs/msg/PoseStamped`
- **Purpose**: Represents robot's position and orientation in space
- **Contains**: Header (timestamp, frame), position (x,y,z), orientation (quaternion)
- **Usage**: Navigation, localization, path planning



Working with ROS2 Messages

1. Message Inspection Commands

```
# Show message structure/definition
ros2 interface show geometry_msgs/msg/PoseStamped
ros2 interface show sensor_msgs/msg/LaserScan

# List all available message types
ros2 interface list | grep msg

# Find messages containing specific text
ros2 interface list | grep -i twist
ros2 interface list | grep -i pose
```

2. Publishing Messages from Command Line

```
# Simple string message
ros2 topic pub /chatter std_msgs/msg/String "data: 'Hello ROS2'"

# TurtleBot movement command (Twist message)
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \
  "linear: {x: 0.2, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.5}"

# Pose with timestamp (realistic robot position)
ros2 topic pub /robot_pose geometry_msgs/msg/PoseStamped \
  "header: {stamp: {sec: 0, nanosec: 0}, frame_id: 'map'},
   pose: {position: {x: 2.0, y: 1.0, z: 0.0},
          orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}}"

# Continuous publishing (--rate flag)
```

Live ROS2 Message Content - What You Actually See

1. TurtleBot3 Pose (Odometry Message)

```
$ ros2 topic echo /odom --once
header:
  stamp:
    sec: 1694123456
    nanosec: 789123456
  frame_id: odom
child_frame_id: base_footprint
pose:
  pose:
    position:
      x: 2.1532423496246338
      y: -0.8765432109876543
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.3826834323650898     # Rotation around Z-axis
      w: 0.9238795325112867     # Quaternion W component
  covariance: [0.01, 0.0, 0.0, 0.0, 0.0, 0.0, ...]
twist:
  twist:
    linear:
      x: 0.15234567890123456    # Forward velocity (m/s)
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
```

More TurtleBot3 Messages - Sensors & Commands

2. Laser Scan Data (LIDAR)

```
$ ros2 topic echo /scan --once
header:
  stamp: {sec: 1694123456, nanosec: 789123456}
  frame_id: base_scan
angle_min: -3.141592653589793     # -180 degrees
angle_max: 3.141592653589793      # +180 degrees
angle_increment: 0.017453292519943295  # ~1 degree
range_min: 0.11999999731779099
range_max: 3.5
ranges: [inf, inf, inf, 2.234, 2.145, 1.987, 1.834, 1.723, ...]
intensities: [0.0, 0.0, 0.0, 47.0, 52.0, 45.0, 38.0, 41.0, ...]
```

3. Velocity Commands (What Teleop Sends)

```
$ ros2 topic echo /cmd_vel --once
linear:
  x: 0.22000000000000003     # Forward speed (m/s)
  y: 0.0                     # Sideways (always 0 for diff drive)
  z: 0.0                     # Up/down (always 0 for ground robot)
angular:
  x: 0.0                     # Roll (always 0)
  y: 0.0                     # Pitch (always 0)
  z: 1.2400000000000002      # Yaw - turning left (+) or right (-)
```

ROS2 Messages & TF2 - Essential Documentation

DOCS: Official ROS2 Documentation Links:

→ **ROS2 Interfaces (Messages, Services, Actions):**

- **Complete Guide:** `https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html`
- Covers message definitions, custom interfaces, and best practices
- Essential for understanding data structures in ROS2

→ **TF2 Transform Framework:**

- **Complete Guide:** `https://docs.ros.org/en/humble/Concepts/Intermediate/About-Tf2.html`
- Coordinate frame management and spatial relationships
- Critical for robot localization and navigation

TIP: Pro Tip: Bookmark these pages! They contain:

- Interactive tutorials with code examples

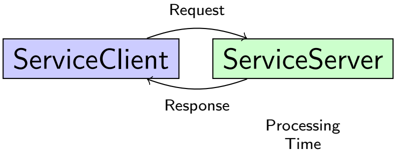
Services: Request-Response Pattern

What are Services?

- **Synchronous communication** - Client waits for response
- **One-to-one interaction** - Single client talks to single server
- **Guaranteed response** - Always get an answer (or timeout)
- **Discrete operations** - Perfect for "do this now" commands
- **Blocking behavior** - Client stops until response received

Real Robot Examples:

- Reset robot position in simulation
- Query current battery level
- Change robot operating mode



Think of it like: Calling a function on a remote computer!

Service Server Example - Python

```python
from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node

class ServiceServer(Node):
    def __init__(self):
        super().__init__('add_two_ints_server')
        self.srv = self.create_service(
            AddTwoInts,
            'add_two_ints',
            self.add_callback)
        self.get_logger().info('Service server ready')

    def add_callback(self, request, response):
        response.sum = request.a + request.b
        self.get_logger().info(f'\\{request.a\\} + \\{request.b\\} = \\{response.sum\\}')
        return response
```

Service Client Example - Python

```python
from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node

class ServiceClient(Node):
    def __init__(self):
        super().__init__('add_two_ints_client')
        self.client = self.create_client(AddTwoInts, 'add_two_ints')
        while not self.client.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Waiting for service...')

    def send_request(self, a, b):
        request = AddTwoInts.Request()
        request.a = a
        request.b = b
        future = self.client.call_async(request)
        return future
```

Custom Service Definition

```
# ComputePath.srv
# Request
geometry_msgs/PoseStamped start
geometry_msgs/PoseStamped goal
float32 planning_time_limit
---
# Response
bool success
nav_msgs/Path path
string error_message
float32 planning_time
```

Service Commands:

```
# List all services
ros2 service list

# Get service type
ros2 service type /service_name

# Call a service
ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 2, b: 3}"
```

TurtleBot3 Services in Action

Real Services with TurtleBot3 Demo:

Gazebo Simulation Services:

```
# Reset robot to starting position
ros2 service call /reset_simulation std_srvs/Empty

# Pause/unpause physics
ros2 service call /pause_physics std_srvs/Empty
ros2 service call /unpause_physics std_srvs/Empty

# Get robot's current pose
ros2 service call /get_model_state gazebo_msgs/
    GetModelState \
  "{model_name: turtlebot3_burger}"

# Teleport robot to new location
ros2 service call /set_entity_state gazebo_msgs/
    SetEntityState \
  "{state: {name: turtlebot3_burger,
   pose: {position: {x: 2.0, y: 1.0}}}}"
```

Robot Configuration Services:

```
# List all available services
ros2 service list | grep turtlebot

# Get service interface info
ros2 service type /scan_matching_status

# Example robot-specific service
ros2 service call /robot_state_publisher \
  /get_loggers example_interfaces/srv/SetParameters
```

Key Insights:

- Services perfect for simulation control
- One-shot operations like reset/teleport
- Query current robot state
- Trigger mode changes instantly

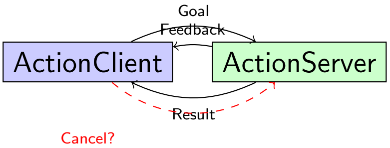
Actions: Long-Running Tasks with Feedback

What are Actions?

- **Asynchronous with feedback** - Start task, get progress updates
- **Cancelable operations** - Can stop task midway if needed
- **Progress monitoring** - Continuous feedback while running
- **Goal-Result-Feedback pattern** - Three-part communication
- **Non-blocking** - Client continues other work while action runs

Perfect for Robot Tasks:

- Navigate to a specific location (takes time!)
- Follow a path with obstacle avoidance
- Manipulator arm movements



Think of it like: Ordering food delivery - you get updates on progress and can cancel if needed!

Action Definition

```
# Fibonacci.action
# Goal
int32 order
---
# Result
int32[] sequence
---
# Feedback
int32[] partial_sequence
```

Action Interface Components:

- **Goal**: What the action should achieve
- **Result**: Final outcome of the action
- **Feedback**: Progress updates during execution
- **Status**: Current state of the action

Action Server Example - Python

```python
import rclpy
from rclpy.action import ActionServer
from rclpy.node import Node
from action_tutorials_interfaces.action import Fibonacci

class FibonacciActionServer(Node):
    def __init__(self):
        super().__init__('fibonacci_action_server')
        self._action_server = ActionServer(
            self, Fibonacci, 'fibonacci',
            self.execute_callback)

    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')
        feedback_msg = Fibonacci.Feedback()
        feedback_msg.partial_sequence = [0, 1]

        for i in range(1, goal_handle.request.order):
            feedback_msg.partial_sequence.append(
                feedback_msg.partial_sequence[i] + feedback_msg.partial_sequence[i-1])
            goal_handle.publish_feedback(feedback_msg)
            # time.sleep(1)  # Simulate processing time

        goal_handle.succeed()
        result = Fibonacci.Result()
        result.sequence = feedback_msg.partial_sequence
        return result
```

TurtleBot3 Navigation Actions

Real Actions with TurtleBot3 Navigation:

Navigation Action Example:

```
# Send navigation goal to TurtleBot3
ros2 action send_goal /navigate_to_pose \
  nav2_msgs/NavigateToPose \
  "{pose: {
    header: {frame_id: map},
    pose: {
      position: {x: 2.0, y: 1.0, z: 0.0},
      orientation: {w: 1.0}
    }
  }}"

# Send goal with feedback monitoring
ros2 action send_goal /navigate_to_pose \
  nav2_msgs/NavigateToPose \
  --feedback \
  "{pose: {header: {frame_id: map},
           pose: {position: {x: -1.0, y: 0.5}}}}"

# List all available actions
ros2 action list

# Get action info
```

What You'll See:

- **Goal**: Target pose coordinates sent
- **Feedback**: Distance remaining, current pose, obstacles detected
- **Result**: Success/failure, final pose, path length
- **Cancellation**: Stop robot mid-navigation if needed

Action vs Service vs Topic:

- **Topic**: Continuous laser data stream
- **Service**: "Reset robot position now!"
- **Action**: "Navigate to kitchen, keep me posted!"

Live Demo Commands:

ROS2 Parameters

What are Parameters?

- Configuration values for nodes
- Dynamic reconfiguration support
- Type-safe (int, double, string, bool, arrays)
- Persistent storage options
- Namespace support

Parameter Features:

- Can be set at launch time
- Can be modified at runtime
- Can have default values
- Can be loaded from YAML files
- Support for parameter events

Using Parameters - Python

```python
import rclpy
from rclpy.node import Node

class ParameterNode(Node):
    def __init__(self):
        super().__init__('parameter_node')

        # Declare parameters with defaults
        self.declare_parameter('robot_name', 'robot1')
        self.declare_parameter('max_speed', 1.0)
        self.declare_parameter('enable_sensors', True)

        # Get parameter values
        robot_name = self.get_parameter('robot_name').value
        max_speed = self.get_parameter('max_speed').value

        # Set parameter callback
        self.add_on_set_parameters_callback(self.parameter_callback)

    def parameter_callback(self, params):
        for param in params:
            if param.name == 'max_speed' and param.value > 5.0:
                return SetParametersResult(successful=False)
        return SetParametersResult(successful=True)
```

Parameter Files (YAML)

```yaml
# config/robot_params.yaml
robot_controller:
  ros__parameters:
    robot_name: "mobile_robot"
    max_speed: 2.0
    max_acceleration: 1.5
    enable_sensors: true

    pid_gains:
      p: 1.0
      i: 0.1
      d: 0.05

    sensor_topics:
      - "/scan"
      - "/odom"
      - "/imu"
```

Parameter Commands

```
# List all parameters for a node
ros2 param list /node_name

# Get parameter value
ros2 param get /node_name parameter_name

# Set parameter value
ros2 param set /node_name parameter_name value

# Load parameters from file
ros2 param load /node_name params.yaml

# Dump parameters to file
ros2 param dump /node_name --output-dir ./

# Run node with parameter file
ros2 run package node --ros-args --params-file config.yaml
```

Launch System Overview

Purpose of Launch Files:

- Start multiple nodes simultaneously
- Configure node parameters
- Set up remappings
- Define node relationships
- Handle complex system configurations
- Support conditional logic

Launch File Formats:

- Python (recommended for ROS2)
- XML (legacy support)
- YAML (simple configurations)

Python Launch File Example

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration

def generate_launch_description():
    # Declare launch arguments
    use_sim_time = LaunchConfiguration('use_sim_time', default='false')

    return LaunchDescription([
        DeclareLaunchArgument(
            'use_sim_time',
            default_value='false',
            description='Use simulation time'),

        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim',
            parameters=[{'use_sim_time': use_sim_time}]),

        Node(
            package='my_package',
            executable='controller',
            name='controller',
            parameters=[{'max_speed': 2.0}],
            remappings=[('/cmd_vel', '/turtle1/cmd_vel')])
    ])
```

Advanced Launch Features

```python
from launch.conditions import IfCondition
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    # Include another launch file
    included_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(get_package_share_directory('nav2_bringup'),
                        'launch', 'navigation_launch.py')),
        launch_arguments={'use_sim_time': 'true'}.items())

    # Conditional node launch
    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        condition=IfCondition(LaunchConfiguration('use_rviz')))

    return LaunchDescription([included_launch, rviz_node])
```

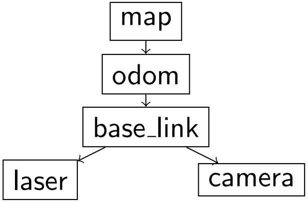
Understanding TF2

What is TF2?

- Transform library for ROS2
- Tracks coordinate frames over time
- Maintains frame relationships
- Handles time synchronization
- Essential for robot navigation

Key Concepts:

- Frames: Coordinate systems
- Transforms: Relationships between frames
- Transform tree: Hierarchy of frames
- Static vs dynamic transforms



Broadcasting Transforms - Python

```python
import rclpy
from rclpy.node import Node
from tf2_ros import TransformBroadcaster
from geometry_msgs.msg import TransformStamped
import time

class FramePublisher(Node):
    def __init__(self):
        super().__init__('tf2_frame_publisher')
        self.tf_broadcaster = TransformBroadcaster(self)
        self.timer = self.create_timer(0.1, self.broadcast_timer_callback)

    def broadcast_timer_callback(self):
        t = TransformStamped()
        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'base_link'
        t.child_frame_id = 'sensor_frame'

        t.transform.translation.x = 0.5
        t.transform.translation.y = 0.0
        t.transform.translation.z = 0.2
        t.transform.rotation.x = 0.0
        t.transform.rotation.y = 0.0
        t.transform.rotation.z = 0.0
        t.transform.rotation.w = 1.0

        self.tf_broadcaster.sendTransform(t)
```

Listening to Transforms - Python

```python
import rclpy
from rclpy.node import Node
from tf2_ros import TransformListener, Buffer
from tf2_ros.transform_listener import TransformException

class FrameListener(Node):
    def __init__(self):
        super().__init__('tf2_frame_listener')
        self.tf_buffer = Buffer()
        self.tf_listener = TransformListener(self.tf_buffer, self)
        self.timer = self.create_timer(1.0, self.timer_callback)

    def timer_callback(self):
        try:
            # Get transform from base_link to sensor_frame
            trans = self.tf_buffer.lookup_transform(
                'base_link', 'sensor_frame',
                rclpy.time.Time())
            self.get_logger().info(f'Transform: \{trans\}')
        except TransformException as ex:
            self.get_logger().info(f'Could not get transform: \{ex\}')
```

ROS2 Visualization Commands:

1. RQt Graph (Node/Topic Visualization)

```
# View the ROS2 computational graph
rqt_graph

# Alternative way to launch
ros2 run rqt_graph rqt_graph
```

WARNING: Troubleshooting rqt_graph:

```
# If rqt_graph fails due to Python version mismatch:
# Issue: Anaconda Python 3.13 vs ROS2 Humble Python 3.10
# Solution: Force Python 3.10 for ROS2 components

export PATH="/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:$PATH"
export PYTHONPATH="/opt/ros/humble/lib/python3.10/site-packages:/opt/ros/humble/local/lib/python3.10/dist-
    packages"

# Verify Python 3.10 is loaded
which python3
python3 --version  # Should show Python 3.10.x

# Source ROS2 environment after PATH changes
source /opt/ros/humble/setup.bash
export ROS_DOMAIN_ID=200

# Now rqt_graph should work
rqt_graph
```

RViz and Demo Commands

4. RViz with TF Display

```
# Launch RViz2 with TF visualization
ros2 run rviz2 rviz2

# Then add "TF" display in RViz to see coordinate frames
```

TurtleBot 3 Demo Commands:

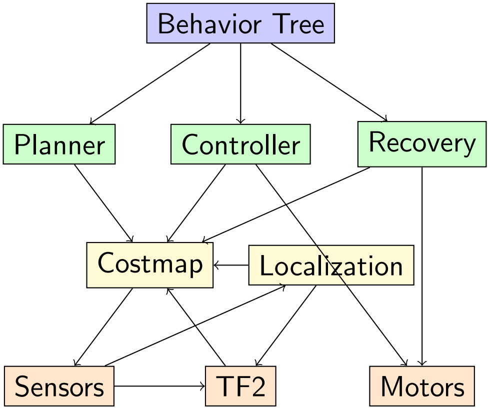
```
# 1. Start your TurtleBot3 demo
./launch_turtlebot3_complete.sh

# 2. In another terminal, show the graph
rqt_graph

# 3. Generate TF tree PDF
ros2 run tf2_tools view_frames

# 4. View all topics and nodes
ros2 topic list
ros2 node list
```

Nav2 Architecture



Nav2 Key Components

- **Planner Server**: Computes paths from A to B
  - NavFn, Smac, Theta*
- **Controller Server**: Follows the computed path
  - DWB, TEB, Regulated Pure Pursuit
- **Recovery Server**: Handles stuck situations
  - Spin, Back Up, Wait
- **Costmap 2D**: Environmental representation
  - Static layer, Obstacle layer, Inflation layer
- **Behavior Trees**: Orchestrates navigation
- **Lifecycle Manager**: Manages node states

Simple Navigation Goal - Python

```python
from nav2_simple_commander.robot_navigator import BasicNavigator
from geometry_msgs.msg import PoseStamped
import rclpy

def main():
    rclpy.init()
    navigator = BasicNavigator()

    # Set initial pose
    initial_pose = PoseStamped()
    initial_pose.header.frame_id = 'map'
    initial_pose.pose.position.x = 0.0
    initial_pose.pose.position.y = 0.0
    navigator.setInitialPose(initial_pose)

    # Wait for Nav2 to activate
    navigator.waitUntilNav2Active()

    # Set goal pose
    goal_pose = PoseStamped()
    goal_pose.header.frame_id = 'map'
    goal_pose.pose.position.x = 2.0
    goal_pose.pose.position.y = 1.0

    # Navigate to goal
    navigator.goToPose(goal_pose)

    while not navigator.isTaskComplete():
        feedback = navigator.getFeedback()
        # Process feedback
```

Common Sensors in ROS2

2D Sensors:

- Laser Scanners (LIDAR)
- Cameras (RGB)
- Ultrasonic sensors
- Infrared sensors

3D Sensors:

- 3D LIDAR
- Depth cameras (RGBD)
- Stereo cameras
- Time-of-Flight cameras

Other Sensors:

- IMU (Inertial Measurement Unit)
- GPS/GNSS
- Wheel encoders
- Force/Torque sensors
- Temperature sensors
- Battery monitors

Processing Laser Scan Data

```
from sensor_msgs.msg import LaserScan
import rclpy
from rclpy.node import Node
import numpy as np

class LaserProcessor(Node):
    def __init__(self):
        super().__init__(’laser_processor’)
        self.subscription = self.create_subscription(
            LaserScan, ’/scan’, self.scan_callback, 10)

    def scan_callback(self, msg):
        # Process laser scan
        ranges = np.array(msg.ranges)

        # Remove invalid readings
        ranges[ranges > msg.range_max] = msg.range_max
        ranges[ranges < msg.range_min] = msg.range_min

        # Find closest obstacle
        min_distance = np.min(ranges)
        min_angle = msg.angle_min + np.argmin(ranges) * msg.angle_increment

        self.get_logger().info(f’Closest obstacle: \{min_distance:.2f\}m at \{np.degrees(min_angle):.1f\} deg’)

        # Check for obstacles in front
        front_angles = len(ranges) // 3
        if np.min(ranges[front_angles:-front_angles]) < 0.5:
            self.get_logger().warn(’Obstacle ahead!’)
```

Image Processing with OpenCV

```python
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import cv2
import rclpy
from rclpy.node import Node

class ImageProcessor(Node):
    def __init__(self):
        super().__init__('image_processor')
        self.bridge = CvBridge()
        self.subscription = self.create_subscription(
            Image, '/camera/image_raw', self.image_callback, 10)
        self.publisher = self.create_publisher(
            Image, '/processed_image', 10)

    def image_callback(self, msg):
        # Convert ROS Image to OpenCV
        cv_image = self.bridge.imgmsg_to_cv2(msg, 'bgr8')

        # Process image (example: edge detection)
        gray = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
        edges = cv2.Canny(gray, 100, 200)

        # Convert back to ROS Image and publish
        processed_msg = self.bridge.cv2_to_imgmsg(edges, 'mono8')
        self.publisher.publish(processed_msg)
```

Point Cloud Processing

```python
from sensor_msgs.msg import PointCloud2
import sensor_msgs.point_cloud2 as pc2
import numpy as np
import rclpy
from rclpy.node import Node

class PointCloudProcessor(Node):
    def __init__(self):
        super().__init__('pointcloud_processor')
        self.subscription = self.create_subscription(
            PointCloud2, '/velodyne_points',
            self.pointcloud_callback, 10)

    def pointcloud_callback(self, msg):
        # Convert PointCloud2 to numpy array
        points = []
        for point in pc2.read_points(msg, field_names=("x", "y", "z"),
                                     skip_nans=True):
            points.append([point[0], point[1], point[2]])

        points = np.array(points)

        # Basic processing
        if len(points) > 0:
            # Ground removal (simple height threshold)
            non_ground = points[points[:, 2] > -0.3]

            # Find closest point
            distances = np.linalg.norm(non_ground, axis=1)
            if len(distances) > 0:
                min_dist = np.min(distances)
                self.get_logger().info(f'Closest point: \{min_dist:.2f\}m')
```

Gazebo Integration with ROS2

Why Simulation?

- Safe testing environment
- No hardware required
- Reproducible scenarios
- Faster development cycles
- Multi-robot testing

Gazebo Features:

- Physics simulation (ODE, Bullet, etc.)
- Sensor simulation
- 3D visualization
- Plugin system for ROS2 integration
- World and model creation tools

URDF Robot Description

```xml
<?xml version="1.0"?>
<robot name="simple_robot">
  <!-- Base Link -->
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.5 0.3 0.1"/>
      </geometry>
      <material name="blue">
        <color rgba="0 0 1 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <box size="0.5 0.3 0.1"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="1.0"/>
      <inertia ixx="0.01" ixy="0.0" ixz="0.0"
               iyy="0.01" iyz="0.0" izz="0.01"/>
    </inertial>
  </link>

  <!-- Wheel Joint -->
  <joint name="wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="wheel"/>
    <origin xyz="0.2 0 -0.05"/>
    <axis xyz="0 1 0"/>
  </joint>
</robot>
```

Launching Gazebo with ROS2

```python
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch_ros.actions import Node
from launch.launch_description_sources import PythonLaunchDescriptionSource
from ament_index_python.packages import get_package_share_directory
import os

def generate_launch_description():
    # Launch Gazebo
    gazebo = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            os.path.join(get_package_share_directory('gazebo_ros'),
                        'launch', 'gazebo.launch.py')]),
        launch_arguments={'world': 'my_world.world'}.items())

    # Spawn robot
    spawn_entity = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=['-entity', 'my_robot',
                   '-topic', 'robot_description'])

    # Robot state publisher
    robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        parameters=[{'robot_description': robot_description}])

    return LaunchDescription([gazebo, spawn_entity, robot_state_publisher])
```

ROS2 Best Practices

Node Design:

- Single responsibility principle
- Use components for reusability
- Implement proper lifecycle management
- Handle exceptions gracefully

Communication:

- Choose appropriate QoS settings
- Use services for configuration
- Use topics for continuous data
- Use actions for long tasks

Performance:

- Minimize message copying
- Use appropriate executor types
- Consider DDS settings
- Profile and optimize bottlenecks

Quality of Service (QoS) Settings

Reliability:

- RELIABLE: No message loss
- BEST_EFFORT: May drop messages

Durability:

- TRANSIENT_LOCAL: Late joiners get history
- VOLATILE: No history for late joiners

History:

- KEEP_LAST: Keep N recent messages
- KEEP_ALL: Keep all messages

Common Profiles:

- Sensor data: Best effort, volatile
- Parameters: Reliable, transient
- Commands: Reliable, volatile

QoS Configuration Example

```python
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy

# Custom QoS for sensor data
sensor_qos = QoSProfile(
    reliability=ReliabilityPolicy.BEST_EFFORT,
    durability=DurabilityPolicy.VOLATILE,
    depth=1
)

# Custom QoS for control commands
control_qos = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,
    durability=DurabilityPolicy.VOLATILE,
    depth=10
)

# Use in publisher/subscriber
self.laser_sub = self.create_subscription(
    LaserScan, '/scan', self.scan_callback,
    qos_profile=sensor_qos)

self.cmd_pub = self.create_publisher(
    Twist, '/cmd_vel',
    qos_profile=control_qos)
```

ROS2 Debugging Tools

Command Line Tools:

- `ros2 doctor`: System diagnostics
- `ros2 wtf`: What The Failure analysis
- `ros2 topic echo`: Monitor messages
- `ros2 param dump`: Save parameters
- `ros2 bag`: Record and replay data

Visualization Tools:

- **RViz2**: 3D visualization
- **rqt**: Plugin-based GUI tools
- **PlotJuggler**: Time series plotting
- **Foxglove Studio**: Web-based viz

Common Debugging Commands

```
# System diagnostics
ros2 doctor --report

# Node introspection
ros2 node info /node_name --verbose

# Topic debugging
ros2 topic echo /topic_name --no-arr  # Don't print arrays
ros2 topic hz /topic_name  # Check frequency
ros2 topic bw /topic_name  # Check bandwidth

# Parameter debugging
ros2 param describe /node_name param_name
ros2 param dump /node_name --print

# Service debugging
ros2 service type /service_name
ros2 service find std_srvs/srv/Empty

# Check node graph
ros2 run rqt_graph rqt_graph

# Record for offline debugging
ros2 bag record -a  # Record all topics
ros2 bag play bag_file.db3 --rate 0.5  # Play at half speed
```

Logging in ROS2

```python
import rclpy
from rclpy.node import Node

class LoggingExample(Node):
    def __init__(self):
        super().__init__('logging_example')

        # Different log levels
        self.get_logger().debug('Debug message')
        self.get_logger().info('Info message')
        self.get_logger().warn('Warning message')
        self.get_logger().error('Error message')
        self.get_logger().fatal('Fatal message')

        # Conditional logging
        self.get_logger().info('Value: %d' % value,
                               throttle_duration_sec=1.0)

        # Logging once
        self.get_logger().info('This logs once', once=True)

# Set log level from command line
# ros2 run pkg node --ros-args --log-level debug
```

Exercise 1: Publisher-Subscriber

Task: Create a temperature monitoring system

1. Create a temperature sensor node that publishes random temperatures
2. Create a monitor node that subscribes and warns if temperature ¿ threshold
3. Add a parameter for the warning threshold
4. Record the data to a bag file

Learning Objectives:

- Understanding pub-sub pattern
- Working with parameters
- Using standard messages
- Data recording

Exercise 2: Service-Based Calculator

Task: Implement a calculator service

1. Define a custom service with operation and two numbers
2. Create a server that performs +, -, *, /
3. Create a client with command-line interface
4. Handle division by zero error

Learning Objectives:

- Creating custom services
- Implementing service servers and clients
- Error handling
- Synchronous communication

Exercise 3: TurtleBot Navigation

Task: Make TurtleBot navigate a square pattern

1. Launch turtlesim
2. Create a node that publishes Twist messages
3. Implement square pattern movement
4. Use TF2 to track turtle position
5. Visualize path in RViz2

Learning Objectives:

- Velocity control
- Working with simulators
- Using TF2
- Visualization tools

Exercise 4: Sensor Fusion

Task: Combine multiple sensor inputs

1. Subscribe to laser scan and odometry
2. Implement obstacle detection from laser
3. Calculate robot velocity from odometry
4. Publish combined status message
5. Create launch file for all nodes

Learning Objectives:

- Multi-topic subscription
- Sensor data processing
- Message synchronization
- Launch file creation

Exercise 5: Action-Based Pick and Place

Task: Simulate a pick-and-place operation

1. Define a PickPlace action (object, location)
2. Create action server with stages (approach, pick, move, place)
3. Send feedback for each stage
4. Implement action client with goal cancellation
5. Add recovery behavior for failures

Learning Objectives:

- Action definition and implementation
- Feedback mechanisms
- Goal cancellation
- State machines

Project 1: Autonomous Line Follower

Components:

- Camera for line detection
- Image processing with OpenCV
- PID controller for steering
- Speed regulation based on line curvature

ROS2 Concepts:

- Image topics and cv_bridge
- Control parameters
- Velocity commands (Twist)
- Launch files for system startup

Extensions:

- Multiple line colors
- Intersection handling
- Traffic sign recognition

Project 2: Multi-Robot Coordination

Components:

- Multiple robot instances
- Centralized task allocator
- Inter-robot communication
- Collision avoidance

ROS2 Concepts:

- Namespaces for multiple robots
- Service-based task allocation
- Distributed system architecture
- TF2 for multi-robot transforms

Extensions:

- Formation control
- Load balancing
- Fault tolerance

Project 3: SLAM and Navigation

Components:

- SLAM using slam_toolbox
- Path planning with Nav2
- Obstacle avoidance
- Goal setting interface

ROS2 Concepts:

- Nav2 stack integration
- Costmap configuration
- Behavior trees
- Map saving/loading

Extensions:

- Multi-floor navigation
- Dynamic obstacle handling
- Semantic mapping

ROS2 Security

Security Features:

- DDS Security plugins
- Authentication
- Access control
- Encryption
- Data integrity

Implementation:

- Generate security certificates
- Configure security policies
- Enable secure DDS
- Audit and monitoring

Real-Time Programming

Real-Time Considerations:

- Deterministic execution
- Priority-based scheduling
- Memory management
- Lock-free programming

ROS2 Real-Time Features:

- Real-time executors
- Static memory allocation
- Priority inheritance
- Deadline monitoring

Micro-ROS

What is Micro-ROS?

- ROS2 for microcontrollers
- Minimal footprint
- RTOS integration
- Bridge to main ROS2 system

Supported Platforms:

- Arduino
- ESP32
- STM32
- Teensy

ROS2 Control

Framework Components:

- Hardware interfaces
- Controllers
- Controller manager
- Transmission interfaces

Common Controllers:

- Joint position/velocity/effort
- Differential drive
- Joint trajectory
- Admittance control

Unit Testing with pytest

```python
import pytest
import rclpy
from my_package.my_node import MyNode

@pytest.fixture
def node():
    rclpy.init()
    node = MyNode()
    yield node
    node.destroy_node()
    rclpy.shutdown()

def test_parameter_validation(node):
    # Test parameter bounds
    node.set_parameters([rclpy.Parameter('speed', value=2.0)])
    assert node.get_parameter('speed').value == 2.0

    # Test invalid parameter
    result = node.set_parameters([rclpy.Parameter('speed', value=10.0)])
    assert not result[0].successful

def test_message_processing(node):
    # Test message handling
    from std\\_msgs.msg import String
    msg = String(data='test')
    node.message_callback(msg)
    assert node.last_message == 'test'
```

Integration Testing

```python
import launch_testing
import unittest
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_test_description():
    # Launch nodes for testing
    return LaunchDescription([
        Node(package='my_package', executable='talker'),
        Node(package='my_package', executable='listener'),
        launch_testing.actions.ReadyToTest()
    ])

class TestCommunication(unittest.TestCase):
    def test_topic_communication(self, proc_output):
        # Wait for expected output
        proc_output.assertWaitFor('Received: Hello', timeout=5)

    def test_service_response(self):
        import rclpy
        from example_interfaces.srv import AddTwoInts

        node = rclpy.create_node('test_client')
        client = node.create_client(AddTwoInts, 'add')

        request = AddTwoInts.Request(a=2, b=3)
        future = client.call_async(request)
        rclpy.spin_until_future_complete(node, future)

        assert future.result().sum == 5
```

Containerization with Docker

Benefits:

- Consistent environment
- Easy deployment
- Version control
- Isolation
- Scalability

Best Practices:

- Multi-stage builds
- Minimal base images
- Layer caching
- Security scanning

#

```
# Install dependencies RUN apt-get update &&
apt-get install -y
python3-pip
ros-humble-nav2
# Copy workspace COPY ./src /ros2_ws/src
# Build RUN cd /ros2_ws&&
colcon build
# Entry point CMD ["ros2", "launch",
    "my_pkg","robot.launch.py"]
```

Continuous Integration/Deployment

CI/CD Pipeline:

1. Code commit triggers build
2. Run unit tests
3. Run integration tests
4. Build Docker image
5. Deploy to robot/cloud

Tools:

- GitHub Actions / GitLab CI
- Industrial CI for ROS
- Ansible for deployment
- Kubernetes for orchestration

System Monitoring

Monitoring Aspects:

- Node health status
- Topic frequencies
- CPU/Memory usage
- Network latency
- Error rates

Tools:

- ros2_monitor
- Prometheus + Grafana
- ELK Stack (Elasticsearch, Logstash, Kibana)
- Custom diagnostics

Learning Resources

Official Documentation:

- `https://docs.ros.org/en/humble/`
- ROS2 Tutorials
- API Documentation
- Design Documents

Books:

- "A Gentle Introduction to ROS2"
- "Programming Robots with ROS"
- "ROS2 in 5 Days" (The Construct)

Online Courses:

- ROS2 Developers Course
- Udemy ROS2 courses
- YouTube tutorials

Community and Support

Forums and Q&A:

- ROS Discourse: `discourse.ros.org`
- ROS Answers: `answers.ros.org`
- Stack Overflow: [ros2] tag
- Reddit: r/ROS

Development:

- GitHub: `github.com/ros2`
- ROS Enhancement Proposals (REPs)
- Working Groups
- ROS World conference

Packages:

- ROS Index: `index.ros.org`
- GitHub ros2 org
- Community packages

Summary

What We Covered:

- ROS2 Architecture
- Installation and Setup
- Nodes and Communication
- Topics, Services, Actions
- Parameters and Launch
- TF2 Transforms
- Navigation Stack
- Perception and Sensors
- Simulation
- Best Practices
- Testing and Debugging
- Deployment

Next Steps:

- Practice with exercises
- Setup a mobile manipulator using ROS2 Humble, Gazebo and RViz2
- Complete Assignment 4 Part 1: `https://kulbir-singh-ahluwalia.com/cs498gc/fa25/assignments.html`

Thank You!

Questions?

CS498GC Mobile Robotics
ROS2 Humble Complete Guide