ROS NOETIC SYSTEM FOR AUTONOMOUS MOBILE ROBOT

ROS (Robot Operating System) is not actually an operating system - it's a flexible framework for writing robot software. It provides services like hardware abstraction, low-level device control, package management, message passing between processes, and tools for visualization and debugging.

As for "noetic" - I should note that if you're referring to something called "neotic edition", I'm not familiar with that term. There may be some confusion here, as the common term in the ROS ecosystem is "Noetic Ninjemys" which is the final ROS 1 distribution released in May 2020.

ROS works primarily on Unix-based platforms (like Ubuntu Linux), though it can run on other operating systems with varying levels of support. It's not a Linux subsystem - rather, it runs on top of an operating system as a collection of tools, libraries, and conventions.

Some key things about ROS:

1. It's organized into "nodes" (executables) that communicate with each other
2. It uses a publisher/subscriber model for communication
3. It has extensive tools for 3D visualization, simulation, and debugging
4. It's widely used in robotics research and development

Let me explain ROS Noetic in more detail.

ROS Noetic Ninjemys (named after the Manning River turtle) is the final release of ROS 1, launched in May 2020. It's designed to work with Ubuntu 20.04 (Focal Fossa) and has several important characteristics:

1. Python 3 Support: Noetic is the first ROS distribution to be Python 3-only, dropping support for Python 2. This was a major change that required many packages to be updated.

2. Key Components:
- RViz: A 3D visualization tool for sensor data and state information

- rqt: A Qt-based framework for GUI development
- Gazebo: Integration with the Gazebo simulator for robot simulation
- catkin: The official build system for ROS packages

3. Package Structure: In Noetic, packages are organized in a workspace, typically called `catkin_ws`, with this structure:

```python
catkin_ws/
    src/
        package_1/
        package_2/
    build/
    devel/
```

4. Basic Concepts:
- Nodes: Individual processes that perform computation
- Topics: Buses over which nodes exchange messages
- Messages: ROS data type used when subscribing or publishing to a topic
- Services: Request/response interactions between nodes

Here's a simple example of a basic ROS Noetic publisher node in Python:

```python
#!/usr/bin/env python3
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz

    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
```

```
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

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

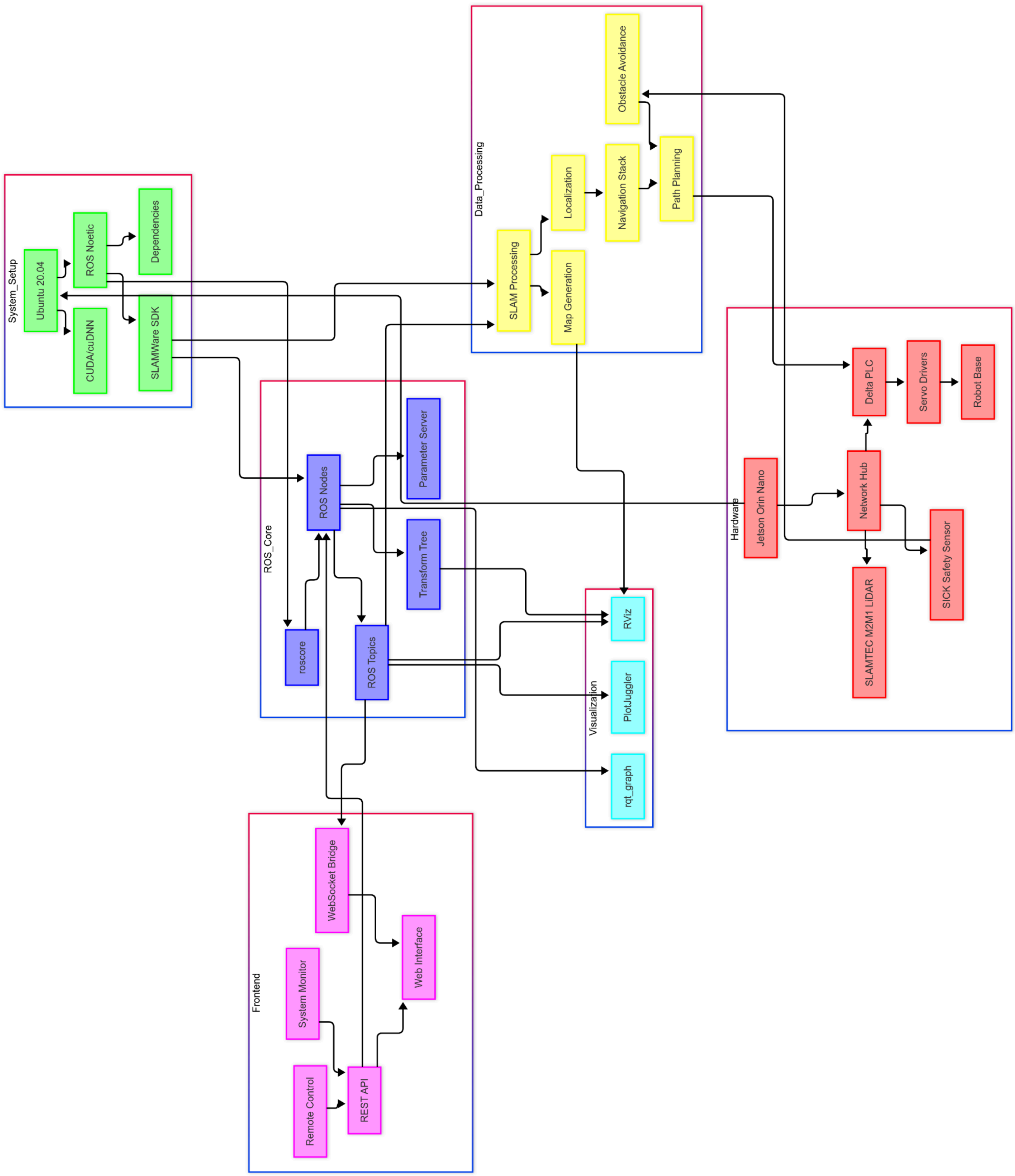
5. Key Features in Noetic:
- Improved navigation stack
- Better manipulation capabilities
- Enhanced MoveIt! integration (robot arm manipulation framework)
- Updated TF2 (transformation) library
- Improved diagnostic tools

6. Development Tools:
- `roscore`: The core ROS system that provides name service
- `rosrun`: Tool to run ROS packages
- `roslaunch`: Tool to launch multiple ROS nodes
- `rostopic`: Command-line tool for ROS topic inspection
- `rqt_graph`: Visual tool for viewing node connections

THINGS NEED TO LEARN

1. How to set up your first ROS Noetic project from scratch
2. How the navigation stack works with robots
3. How to work with ROS topics, services, and actions in detail
4. How to use simulation tools like Gazebo with ROS Noetic
5. How to integrate sensors and work with sensor data
6. Understanding the message passing system and communication between nodes
7. Working with transforms (TF2) for robot coordination
8. How to use visualization tools like RViz effectively

SYSTEM WORKFLOW



1. System Integration Flow:

````
```
LiDAR Hardware → Slamware SDK → ROS Nodes → RViz Visualization
```
````

2. Component Roles:

- LiDAR (Light Detection and Ranging):
  - Provides real-time distance measurements and point cloud data
  - Communicates raw sensor data through serial/USB connection
  - Typically outputs 360-degree scans of the environment

- Slamware SDK:
  - Acts as middleware between LiDAR hardware and ROS
  - Handles SLAM (Simultaneous Localization and Mapping)
  - Processes raw LiDAR data into usable mapping information
  - Provides APIs for robot control and sensor data access

- ROS Noetic Framework:
  - Creates nodes to handle different functionalities:
    - `/scan` topic for LiDAR data
    - `/map` topic for generated maps
    - `/tf` transforms for robot positioning
  - Manages communication between components
  - Handles data processing and distribution

- RViz:
  - Visualizes all the processed data in real-time
  - Displays:
    - LiDAR scan points
    - Generated map
    - Robot position and orientation
    - Navigation paths and plans

3. Basic Setup Flow:

```python
# 1. Install required packages
sudo apt-get install ros-noetic-rviz
sudo apt-get install ros-noetic-navigation

# 2. Create catkin workspace
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make

# 3. Install Slamware SDK
# (Follow Slamware SDK installation instructions for your
specific LiDAR)
```

4. Typical Launch Sequence:

```bash
# 1. Start ROS core
roscore

# 2. Launch Slamware ROS node
roslaunch slamware_ros_sdk slamware_ros_sdk_server_node.launch

# 3. Launch RViz
rosrun rviz rviz -d `rospack find
slamware_ros_sdk`/rviz/slamware_ros_sdk_server_node.rviz
```

5. Data Flow:

```
LiDAR → Slamware SDK → ROS Topics → RViz
              ↓
         Map Generation
              ↓
      Position Estimation
              ↓
Navigation Planning (optional)
```

6. Common ROS Topics Used:

```
/scan           # Raw LiDAR data
/map            # Generated map data
/tf             # Transform data
/cmd_vel        # Robot velocity commands
/odom           # Odometry information
```

7. RViz Configuration:

- Add displays for:
  - LaserScan (for raw LiDAR data)
  - Map (for SLAM-generated map)
  - TF (for coordinate transforms)
  - Robot Model (if using URDF)

8. Common Issues and Solutions:

- LiDAR Connection:

````
```bash
  # Check USB permissions
  sudo usermod -a -G dialout $USER
  # Verify device connection
  ls -l /dev/ttyUSB*
  ```
````

- Slamware SDK Issues:

````
```bash
  # Check if SDK node is running
  rostopic list | grep slamware
  # Check SDK logs
  rosservice call /slamware_ros_sdk_server_node/get_status
  ```
````

Let me explain the key components and their interactions:

1. Network Configuration:

```bash
# LAN Hub Network Settings
LIDAR IP: [Configure SLAMTEC M2M1 IP]
PLC IP: [Configure Delta PLC IP]
SICK Sensor IP: [Configure SICK IP]
Ubuntu PC IP: [Configure Static IP]
```

2. ROS Package Setup:

```bash
# Create and configure workspace
mkdir -p ~/robot_ws/src
cd ~/robot_ws/src

# Clone necessary packages
git clone [slamware_ros_sdk_url]
git clone [sick_safetyscanners_url]
git clone [differential_drive_controller_url]

# Build workspace
cd ~/robot_ws
catkin_make
```

3. Launch File Structure:

```
robot_ws/
├── src/
│   ├── robot_bringup/
│   │   └── launch/
│   │       ├── bringup.launch    # Main launch file
│   │       ├── slam.launch       # SLAM configuration
│   │       ├── navigation.launch # Navigation stack
│   │       └── safety.launch     # SICK sensor configuration
```

4. Main Launch File (bringup.launch):

```xml
<launch>
    <!-- SLAMTEC M2M1 LiDAR -->
    <include file="$(find
slamware_ros_sdk)/launch/slamware_ros_sdk_server_node.launch">
        <arg name="ip_address" value="[LIDAR_IP]"/>
    </include>

    <!-- SICK Safety Scanner -->
    <include file="$(find
sick_safetyscanners)/launch/sick_scanner.launch">
        <arg name="hostname" value="[SICK_IP]"/>
    </include>

    <!-- Differential Drive Controller -->
    <include file="$(find
differential_drive_controller)/launch/controller.launch"/>
```

```
    <!-- Navigation Stack -->
    <include file="$(find
robot_navigation)/launch/move_base.launch"/>

    <!-- RViz -->
    <node name="rviz" pkg="rviz" type="rviz"
          args="-d $(find
robot_bringup)/rviz/robot_config.rviz"/>
</launch>
```

5. PLC Communication Node:

```python
#!/usr/bin/env python3
import rospy
from geometry_msgs.msg import Twist
import socket

class PLCController:
    def __init__(self):
        self.plc_ip = "[PLC_IP]"
        self.plc_port = 502  # Modbus TCP port
        self.sub = rospy.Subscriber('/cmd_vel', Twist,
self.cmd_vel_callback)

    def cmd_vel_callback(self, msg):
        # Convert ROS velocity commands to PLC format
        left_wheel_vel = msg.linear.x - msg.angular.z
        right_wheel_vel = msg.linear.x + msg.angular.z
        # Send to PLC via Modbus TCP
        self.send_to_plc(left_wheel_vel, right_wheel_vel)
```

6. Safety Integration:

```python
def safety_callback(self, msg):
    if msg.protective_field_violated:
        # Emergency stop via PLC
        self.send_emergency_stop()
        rospy.logwarn("Safety field violated - Emergency Stop")
```

7. System Startup Sequence:

```bash
# 1. Start ROS core
roscore

# 2. Launch main system
roslaunch robot_bringup bringup.launch

# 3. Enable motors via PLC
rosservice call /enable_motors

# 4. Start autonomous navigation (if needed)
rosservice call /start_navigation
```

8. RViz Configuration:

- Add displays for:
  - LaserScan (LiDAR data)
  - Map
  - Robot Model
  - Safety Fields
  - Navigation Path
  - Obstacles

How to set up your first ROS Noetic project from scratch

1. First, create a ROS workspace:

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

# Initialize the workspace
catkin_make

# Source the workspace
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```
````

2. Create a new ROS package:

````
```bash
cd ~/catkin_ws/src
# Syntax: catkin_create_pkg <package_name> <depend1> <depend2> ...
catkin_create_pkg my_first_project rospy std_msgs geometry_msgs
```
````

3. Package Structure:

````
```

my_first_project/
├── CMakeLists.txt
├── package.xml
├── src/
````
````

```
├── scripts/
├── launch/
└── config/
```

4. Edit package.xml to add dependencies:

```xml
<?xml version="1.0"?>
<package format="2">
  <name>my_first_project</name>
  <version>0.0.1</version>
  <description>My first ROS project</description>
  <maintainer email="your@email.com">Your Name</maintainer>
  <license>BSD</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>rospy</build_export_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
</package>
```

5. Create a simple publisher node (scripts/publisher.py):

```python
#!/usr/bin/env python3
import rospy
from std_msgs.msg import String

def publisher():
    # Initialize the node
```

````
    rospy.init_node('my_publisher', anonymous=True)
    # Create a publisher object
    pub = rospy.Publisher('my_topic', String, queue_size=10)
    # Set the publishing rate
    rate = rospy.Rate(1) # 1 Hz

    while not rospy.is_shutdown():
        message = "Hello ROS! Time: %s" % rospy.get_time()
        rospy.loginfo(message)
        pub.publish(message)
        rate.sleep()

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

6. Create a simple subscriber node (scripts/subscriber.py):

````
```python
#!/usr/bin/env python3
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo("Received: %s", data.data)

def subscriber():
    # Initialize the node
    rospy.init_node('my_subscriber', anonymous=True)
    # Subscribe to the topic
    rospy.Subscriber('my_topic', String, callback)
    # Keep the node running
````

```
    rospy.spin()

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

7. Make the Python scripts executable:

```bash
chmod +x scripts/publisher.py
chmod +x scripts/subscriber.py
```

8. Create a launch file (launch/my_project.launch):

```xml
<launch>
    <node name="publisher_node" pkg="my_first_project"
type="publisher.py" output="screen"/>
    <node name="subscriber_node" pkg="my_first_project"
type="subscriber.py" output="screen"/>
</launch>
```

9. Build the workspace:

```bash
cd ~/catkin_ws
catkin_make
```

10. Run your project:

````
```bash
# Method 1: Run nodes individually
roscore
rosrun my_first_project publisher.py
rosrun my_first_project subscriber.py

# Method 2: Use launch file
roslaunch my_first_project my_project.launch
```
````

11. Test your system:

````
```bash
# Open new terminal and check:
rostopic list  # Shows available topics
rostopic echo /my_topic  # Shows messages on the topic
rqt_graph  # Shows node graph visualization
```
````

Common debugging commands:

````
```bash
roswtf  # Checks for common problems
rostopic hz /my_topic  # Shows publishing rate
rosnode info /publisher_node  # Shows node information
```
````

Tips for development:

1. Always make Python scripts executable (`chmod +x`)
2. Remember to `catkin_make` after changes
3. Use `roscd my_first_project` to quickly navigate to your package
4. Use `rqt_console` to monitor node outputs and errors
5. Keep your code organized in appropriate directories (src/, scripts/, launch/, config/)

How the navigation stack works with robots

1. **Setting up Costmaps**:

```yaml
# costmap_common_params.yaml
global_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 5.0
  static_map: true

  # Obstacle marking parameters
  obstacle_range: 2.5
  raytrace_range: 3.0
  inflation_radius: 0.55
  transform_tolerance: 0.5

  # Layers
  plugins:
    - {name: static_layer, type: "costmap_2d::StaticLayer"}
    - {name: obstacle_layer, type: "costmap_2d::ObstacleLayer"}
    - {name: inflation_layer, type:
"costmap_2d::InflationLayer"}
```

Example implementation:

```xml
<!-- costmap_setup.launch -->
<launch>
    <node name="costmap_node" pkg="costmap_2d"
type="costmap_2d_node">
        <rosparam file="$(find
my_robot)/config/costmap_common_params.yaml" command="load"/>
```

```xml
        <rosparam file="$(find my_robot)/config/local_costmap_params.yaml" command="load"/>
        <rosparam file="$(find my_robot)/config/global_costmap_params.yaml" command="load"/>
    </node>
</launch>
```

2. **Tuning Navigation Parameters**:

```yaml
# move_base_params.yaml
move_base:
  # Planning parameters
  planner_frequency: 1.0
  planner_patience: 5.0

  # Controller parameters
  controller_frequency: 20.0
  controller_patience: 15.0

  # Recovery behaviors
  recovery_behavior_enabled: true
  clearing_rotation_allowed: true

  # Goal tolerance
  xy_goal_tolerance: 0.10
  yaw_goal_tolerance: 0.05
```

3. **Implementing Custom Behaviors**:

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

```
import rospy
from move_base_msgs.msg import MoveBaseAction
from geometry_msgs.msg import Pose
import actionlib

class CustomNavigationBehavior:
    def __init__(self):
        self.move_base_client =
actionlib.SimpleActionClient('move_base', MoveBaseAction)
        self.move_base_client.wait_for_server()

    def execute_custom_behavior(self):
        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = "map"
        goal.target_pose.header.stamp = rospy.Time.now()

        # Custom behavior implementation
        goal.target_pose.pose = self.calculate_next_pose()
        self.move_base_client.send_goal(goal)

    def calculate_next_pose(self):
        # Custom path planning logic
        pass
```

4. **Path Planning Algorithms**:

```python
# Custom global planner implementation
class CustomGlobalPlanner(nav_core.BaseGlobalPlanner):
    def __init__(self):
        rospy.init_node('custom_planner')

    def make_plan(self, start, goal):
```

```
        path = []

        # Implement A* algorithm
        open_list = [(start, 0)]
        closed_list = set()

        while open_list:
            current, cost = open_list.pop(0)
            if self.is_goal(current, goal):
                return self.reconstruct_path(current)

            for neighbor in self.get_neighbors(current):
                if neighbor not in closed_list:
                    open_list.append((neighbor, cost + 1))

            closed_list.add(current)

        return path
```

5. **Recovery Behaviors**:

```yaml
# recovery_behaviors.yaml
recovery_behaviors:
  - name: conservative_reset
    type: clear_costmap_recovery/ClearCostmapRecovery
  - name: aggressive_reset
    type: clear_costmap_recovery/ClearCostmapRecovery
  - name: rotate_recovery
    type: rotate_recovery/RotateRecovery
```

Implementation:

```python
class CustomRecoveryBehavior:
    def __init__(self):
        self.recovery_behaviors = []
        self.load_recovery_behaviors()

    def execute_recovery(self):
        for behavior in self.recovery_behaviors:
            try:
                behavior.runBehavior()
                if self.check_recovery_success():
                    return True
            except:
                rospy.logwarn("Recovery behavior failed")
        return False
```

6. **Integration with Sensors**:

```python
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import OccupancyGrid

class SensorIntegration:
    def __init__(self):
        self.laser_sub = rospy.Subscriber('scan', LaserScan,
self.laser_callback)
        self.costmap_pub =
rospy.Publisher('local_costmap/updates', OccupancyGrid,
queue_size=1)
```

```python
    def laser_callback(self, scan):
        # Process laser scan data
        ranges = scan.ranges
        # Update costmap based on laser data
        costmap = self.update_costmap(ranges)
        self.costmap_pub.publish(costmap)

    def update_costmap(self, ranges):
        costmap = OccupancyGrid()
        # Convert laser readings to costmap cells
        # Implementation of sensor data integration
        return costmap
```

Configuration for sensor integration:

```yaml
# sensor_params.yaml
laser_scan_sensor:
  sensor_frame: laser
  data_type: LaserScan
  topic: scan
  marking: true
  clearing: true
  min_obstacle_height: 0.05
  max_obstacle_height: 0.35
```

How to work with ROS topics, services, and actions in detail

1. **Topics in Detail**:

- Think of topics as one-way data streams
- Multiple publishers/subscribers can use the same topic
- Best for continuous data flow (like sensor readings, robot state)

Example with custom message type:

````
```python
# First create custom message (my_msgs/msg/RobotState.msg)
string robot_name
float64 battery_level
geometry_msgs/Pose current_pose

# Publisher with custom message
#!/usr/bin/env python3
import rospy
from my_msgs.msg import RobotState
from geometry_msgs.msg import Pose

class DetailedPublisher:
    def __init__(self):
        rospy.init_node('detailed_publisher')
        self.pub = rospy.Publisher('robot_state', RobotState,
queue_size=10)
        self.rate = rospy.Rate(1)

    def publish_state(self):
        msg = RobotState()
        msg.robot_name = "Robot1"
        msg.battery_level = 75.5
        msg.current_pose = Pose()  # Fill pose data
        self.pub.publish(msg)
```
````

2. **Services in Detail**:

- Best for request/response interactions
- Blocking calls (waits for response)
- Good for computational tasks or queries

Example of a custom service:

````
```python
# Custom service definition (my_msgs/srv/RobotCommand.srv)
string command
float64[] parameters
---
bool success
string message
float64[] result

# Service server with error handling
class DetailedServiceServer:
    def __init__(self):
        rospy.init_node('detailed_server')
        self.service = rospy.Service(
            'robot_command',
            RobotCommand,
            self.handle_command
        )

    def handle_command(self, req):
        response = RobotCommandResponse()
        try:
            # Validate command
            if req.command not in ['move', 'stop', 'reset']:
                raise ValueError("Invalid command")

            # Process command
            if req.command == 'move':
````
````

```
                success, result =
self.process_move(req.parameters)
                response.success = success
                response.result = result

            return response

        except Exception as e:
            response.success = False
            response.message = str(e)
            return response
```

3. **Actions in Detail**:

- Best for long-running tasks
- Non-blocking (can be cancelled)
- Provides continuous feedback

Example of a complex action:

```python
# Custom action definition (my_msgs/action/RobotTask.action)
# Goal
string task_type
float64[] parameters
---
# Result
bool success
string message
float64[] final_state
---
# Feedback
float64 percent_complete
```

```
string current_state
float64[] current_parameters

# Detailed action server
class DetailedActionServer:
    def __init__(self):
        self._action_name = "robot_task"
        self._as = actionlib.SimpleActionServer(
            self._action_name,
            RobotTaskAction,
            execute_cb=self.execute_cb,
            auto_start=False
        )
        self._as.register_preempt_callback(self.preempt_cb)
        self._as.start()

    def execute_cb(self, goal):
        success = True
        feedback = RobotTaskFeedback()
        result = RobotTaskResult()

        # Set rate for feedback
        rate = rospy.Rate(1)

        try:
            for i in range(100):
                # Check for preempt (cancel) request
                if self._as.is_preempt_requested():
                    self._as.set_preempted()
                    success = False
                    break

                # Update feedback
                feedback.percent_complete = i + 1
                feedback.current_state = f"Processing step
{i+1}"
```

```python
                feedback.current_parameters = [i + 1]
                self._as.publish_feedback(feedback)

                rate.sleep()

            if success:
                result.success = True
                result.message = "Task completed successfully"
                result.final_state = [100]
                self._as.set_succeeded(result)

        except Exception as e:
            result.success = False
            result.message = str(e)
            self._as.set_aborted(result)

    def preempt_cb(self):
        rospy.loginfo(f"{self._action_name}: Preempted")
```

4. **Advanced Topic Techniques**:

```python
class AdvancedTopics:
    def __init__(self):
        # Latching publisher (keeps last message)
        self.latched_pub = rospy.Publisher(
            'latched_topic',
            String,
            queue_size=1,
            latch=True
        )

        # Subscriber with queue size and callback args
        self.sub = rospy.Subscriber(
            'complex_topic',
            String,
            callback=self.callback,
            callback_args={'custom_arg': 'value'},
            queue_size=10
        )

        # Topic with custom transport hints
        self.tcp_pub = rospy.Publisher(
            'reliable_topic',
            String,
            queue_size=10,
            transport_hints={'tcp_nodelay': True}
        )
```

How to use simulation tools like Gazebo with ROS Noetic

1. **Basic Gazebo-ROS Integration**:

````
```bash
# Install necessary packages
sudo apt-get install ros-noetic-gazebo-ros-pkgs
ros-noetic-gazebo-ros-control
```
````

2. **Robot Model Setup (URDF)**:

````
```xml
<?xml version="1.0"?>
<robot name="my_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 0.8 1"/>
            </material>
        </visual>
        <collision>
            <geometry>
                <box size="0.5 0.3 0.1"/>
            </geometry>
        </collision>
        <inertial>
            <mass value="5.0"/>
            <inertia ixx="0.1" ixy="0" ixz="0" iyy="0.1" iyz="0"
izz="0.1"/>
        </inertial>
````
````

```xml
    </link>

    <!-- Add sensors -->
    <gazebo reference="base_link">
        <sensor type="ray" name="lidar">
            <pose>0 0 0.1 0 0 0</pose>
            <visualize>true</visualize>
            <update_rate>40</update_rate>
            <ray>
                <scan>
                    <horizontal>
                        <samples>720</samples>
                        <resolution>1</resolution>
                        <min_angle>-1.570796</min_angle>
                        <max_angle>1.570796</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <min>0.10</min>
                    <max>30.0</max>
                    <resolution>0.01</resolution>
                </range>
                <noise>
                    <type>gaussian</type>
                    <mean>0.0</mean>
                    <stddev>0.01</stddev>
                </noise>
            </ray>
            <plugin name="gazebo_ros_head_hokuyo_controller"
filename="libgazebo_ros_laser.so">
                <topicName>/scan</topicName>
                <frameName>lidar_link</frameName>
            </plugin>
        </sensor>
    </gazebo>
</robot>
```

````
```
````

3. **Launch File Setup**:

````
```xml
<launch>
    <!-- Start Gazebo with empty world -->
    <include file="$(find
gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find
my_robot_gazebo)/worlds/my_world.world"/>
        <arg name="paused" value="false"/>
        <arg name="use_sim_time" value="true"/>
        <arg name="gui" value="true"/>
        <arg name="recording" value="false"/>
        <arg name="debug" value="false"/>
    </include>

    <!-- Load robot description -->
    <param name="robot_description"
           command="$(find xacro)/xacro --inorder '$(find
my_robot_description)/urdf/robot.urdf.xacro'" />

    <!-- Spawn robot -->
    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
          args="-param robot_description -urdf -model my_robot"
/>

    <!-- Robot state publisher -->
    <node pkg="robot_state_publisher"
type="robot_state_publisher" name="robot_state_publisher">
        <param name="publish_frequency" type="double"
value="50.0" />
    </node>
</launch>
````

````
```
````

4. **Custom World Creation**:

````
```xml
<?xml version="1.0" ?>
<sdf version="1.5">
    <world name="my_world">
        <!-- Physics settings -->
        <physics type="ode">
            <max_step_size>0.001</max_step_size>
            <real_time_factor>1</real_time_factor>
            <real_time_update_rate>1000</real_time_update_rate>
            <gravity>0 0 -9.81</gravity>
        </physics>

        <!-- Lighting -->
        <include>
            <uri>model://sun</uri>
        </include>

        <!-- Ground plane -->
        <include>
            <uri>model://ground_plane</uri>
        </include>

        <!-- Custom obstacles -->
        <model name="box1">
            <pose>2 2 0.5 0 0 0</pose>
            <static>true</static>
            <link name="link">
                <collision name="collision">
                    <geometry>
                        <box>
                            <size>1 1 1</size>
````

```
                        </box>
                    </geometry>
                </collision>
                <visual name="visual">
                    <geometry>
                        <box>
                            <size>1 1 1</size>
                        </box>
                    </geometry>
                </visual>
            </link>
        </model>
    </world>
</sdf>
```

5. **Robot Control Plugin**:

```cpp
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/common/common.hh>
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>

namespace gazebo {
    class RobotController : public ModelPlugin {
        private:
            physics::ModelPtr model;
            event::ConnectionPtr updateConnection;
            ros::NodeHandle* node;
            ros::Subscriber cmdVelSub;

        public:
            void Load(physics::ModelPtr _parent, sdf::ElementPtr
```

```
_sdf) {
                model = _parent;

                // Initialize ROS
                if (!ros::isInitialized()) {
                    int argc = 0;
                    char **argv = NULL;
                    ros::init(argc, argv,
"gazebo_robot_controller");
                }
                node = new ros::NodeHandle();

                // Subscribe to command velocity
                cmdVelSub = node->subscribe("/cmd_vel", 1,
                    &RobotController::OnCmdVel, this);

                // Update loop
                updateConnection =
event::Events::ConnectWorldUpdateBegin(
                    boost::bind(&RobotController::OnUpdate,
this, _1));
            }

            void OnCmdVel(const geometry_msgs::Twist::ConstPtr&
msg) {
                // Apply velocities to the model
                model->SetLinearVel(
                    ignition::math::Vector3d(msg->linear.x,
msg->linear.y, 0));
                model->SetAngularVel(
                    ignition::math::Vector3d(0, 0,
msg->angular.z));
            }

            void OnUpdate(const common::UpdateInfo & /*_info*/)
{
```

```
                // Periodic update code
            }
    };
    GZ_REGISTER_MODEL_PLUGIN(RobotController)
}
```

How to integrate sensors and work with sensor data

1. **LiDAR Integration**:

```python
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import LaserScan
import numpy as np

class LidarProcessor:
    def __init__(self):
        rospy.init_node('lidar_processor')
        self.scan_sub = rospy.Subscriber('/scan', LaserScan, self.scan_callback)
        self.processed_pub = rospy.Publisher('/processed_scan', LaserScan, queue_size=10)

    def scan_callback(self, scan_msg):
        # Access scan data
        ranges = np.array(scan_msg.ranges)
        angles = np.arange(scan_msg.angle_min,
                         scan_msg.angle_max + scan_msg.angle_increment,
                         scan_msg.angle_increment)

        # Filter invalid readings
        valid_ranges = ranges[np.isfinite(ranges)]
        valid_angles = angles[np.isfinite(ranges)]

        # Process data (example: remove outliers)
        mean = np.mean(valid_ranges)
        std = np.std(valid_ranges)
        filtered_ranges = valid_ranges[abs(valid_ranges - mean) < 2 * std]
```

````
```
````

2. **Camera Integration**:

````
```python
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import cv2

class CameraProcessor:
    def __init__(self):
        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber('/camera/image_raw',
Image,
                                         self.image_callback)
        self.processed_pub = rospy.Publisher('/processed_image',
Image,
                                            queue_size=10)

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

            # Image processing
            processed = cv2.GaussianBlur(cv_image, (5, 5), 0)
            edges = cv2.Canny(processed, 100, 200)

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

````
        except Exception as e:
            rospy.logerr(f"Error processing image: {e}")
```
````

3. **IMU Integration**:

````
```python
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import Imu
from geometry_msgs.msg import Vector3
import tf.transformations as tf

class ImuProcessor:
    def __init__(self):
        rospy.init_node('imu_processor')
        self.imu_sub = rospy.Subscriber('/imu/data', Imu,
self.imu_callback)
        self.orientation_pub =
rospy.Publisher('/robot_orientation',
                                             Vector3,
queue_size=10)

    def imu_callback(self, imu_msg):
        # Extract quaternion
        quaternion = (
            imu_msg.orientation.x,
            imu_msg.orientation.y,
            imu_msg.orientation.z,
            imu_msg.orientation.w
        )

        # Convert to Euler angles
        euler = tf.euler_from_quaternion(quaternion)
````

```
        # Publish roll, pitch, yaw
        orientation = Vector3()
        orientation.x = euler[0]  # roll
        orientation.y = euler[1]  # pitch
        orientation.z = euler[2]  # yaw
        self.orientation_pub.publish(orientation)
```

4. **Multi-Sensor Fusion**:

```python
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import LaserScan, Imu
from nav_msgs.msg import Odometry
import message_filters

class SensorFusion:
    def __init__(self):
        # Synchronize multiple sensors
        self.laser_sub = message_filters.Subscriber('/scan',
LaserScan)
        self.imu_sub = message_filters.Subscriber('/imu/data',
Imu)
        self.odom_sub = message_filters.Subscriber('/odom',
Odometry)

        # Time synchronizer
        ts = message_filters.ApproximateTimeSynchronizer(
            [self.laser_sub, self.imu_sub, self.odom_sub],
            queue_size=10,
            slop=0.1  # 100ms tolerance
        )
        ts.registerCallback(self.sync_callback)
```

```
    def sync_callback(self, scan_msg, imu_msg, odom_msg):
        # Process synchronized data
        try:
            # Combine sensor data for improved state estimation
            position = odom_msg.pose.pose.position
            orientation = imu_msg.orientation
            obstacle_ranges = scan_msg.ranges

            # Fusion algorithm implementation
            self.process_sensor_data(position, orientation,
obstacle_ranges)

        except Exception as e:
            rospy.logerr(f"Fusion error: {e}")

    def process_sensor_data(self, position, orientation,
ranges):
        # Implementation of sensor fusion algorithm
        pass
```

5. **Custom Sensor Integration**:

```python
#!/usr/bin/env python3
import rospy
import serial
from std_msgs.msg import Float32MultiArray

class CustomSensorDriver:
    def __init__(self):
        rospy.init_node('custom_sensor_driver')
        self.serial_port = serial.Serial(
            port='/dev/ttyUSB0',
            baudrate=115200,
```

```
            timeout=1.0
        )
        self.sensor_pub = rospy.Publisher('custom_sensor_data',
                                          Float32MultiArray,
queue_size=10)
        self.rate = rospy.Rate(10)  # 10 Hz

    def read_sensor(self):
        while not rospy.is_shutdown():
            try:
                # Read from serial port
                if self.serial_port.in_waiting:
                    data =
self.serial_port.readline().decode('utf-8').strip()
                    values = [float(x) for x in data.split(',')]

                    # Publish data
                    msg = Float32MultiArray(data=values)
                    self.sensor_pub.publish(msg)

            except Exception as e:
                rospy.logerr(f"Error reading sensor: {e}")

            self.rate.sleep()
```

Understanding the message passing system and communication between nodes

1. **Basic Message Passing Architecture**:

```
```python
#!/usr/bin/env python3
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry

class CommunicationExample:
    def __init__(self):
        rospy.init_node('communication_node')

        # Publishers
        self.cmd_pub = rospy.Publisher('/cmd_vel', Twist,
queue_size=10)
        self.status_pub = rospy.Publisher('/robot_status',
String, queue_size=10)

        # Subscribers
        self.odom_sub = rospy.Subscriber('/odom', Odometry,
self.odom_callback)

        # Message buffering
        self.latest_odom = None
        self.rate = rospy.Rate(10)  # 10 Hz

    def odom_callback(self, msg):
        self.latest_odom = msg
        self.process_and_forward()
```
```

```
    def process_and_forward(self):
        if self.latest_odom is not None:
            # Create new command based on odometry
            cmd = Twist()
            cmd.linear.x = 0.5  # Example velocity
            self.cmd_pub.publish(cmd)
```
```

2. **Advanced Communication Patterns**:

```
```python
from std_msgs.msg import Float64MultiArray
import message_filters

class AdvancedCommunication:
    def __init__(self):
        # Synchronized subscribers
        self.laser_sub = message_filters.Subscriber('/scan', 
LaserScan)
        self.odom_sub = message_filters.Subscriber('/odom', 
Odometry)

        # Time Synchronizer
        ts = message_filters.TimeSynchronizer(
            [self.laser_sub, self.odom_sub],
            queue_size=10
        )
        ts.registerCallback(self.sync_callback)

        # Dynamic publisher
        self.dynamic_pub = rospy.Publisher(
            '/dynamic_topic',
            Float64MultiArray,
            queue_size=10,
```

```
            tcp_nodelay=True  # Reduced latency
        )

    def sync_callback(self, laser_msg, odom_msg):
        # Process synchronized messages
        processed_data = self.process_sync_data(laser_msg, 
odom_msg)
        self.dynamic_pub.publish(processed_data)
```

3. **Custom Message Types**:

```python
# In custom_msgs/msg/RobotState.msg
Header header
geometry_msgs/Pose pose
float64 battery_level
string[] active_controllers
bool[] sensor_status

# Using custom messages
from custom_msgs.msg import RobotState

class CustomMessageHandler:
    def __init__(self):
        self.state_pub = rospy.Publisher(
            '/robot_state',
            RobotState,
            queue_size=10
        )

    def publish_state(self):
        msg = RobotState()
        msg.header.stamp = rospy.Time.now()
        msg.battery_level = 0.75
```

```
        msg.active_controllers = ['navigation', 'manipulation']
        msg.sensor_status = [True, True, False]
        self.state_pub.publish(msg)
```

4. **Error Handling and Recovery**:

```python
class RobustCommunication:
    def __init__(self):
        self.retry_count = 3
        self.retry_delay = rospy.Duration(1.0)

        try:
            self.setup_communications()
        except rospy.ROSException as e:
            rospy.logerr(f"Failed to initialize: {e}")
            self.attempt_recovery()

    def setup_communications(self):
        self.pub = rospy.Publisher('/topic', String,
queue_size=10)
        rospy.wait_for_service('required_service', timeout=5.0)

    def attempt_recovery(self):
        for attempt in range(self.retry_count):
            try:
                rospy.sleep(self.retry_delay)
                self.setup_communications()
                rospy.loginfo("Recovery successful")
                break
            except rospy.ROSException as e:
                rospy.logwarn(f"Recovery attempt {attempt + 1}
failed: {e}")
```

5. **Latched Topics and Persistent Communication**:

```python
class PersistentCommunication:
    def __init__(self):
        # Latched publisher (maintains last message)
        self.config_pub = rospy.Publisher(
            '/robot_config',
            String,
            queue_size=1,
            latch=True
        )

        # Subscriber with persistent connection
        self.persistent_sub = rospy.Subscriber(
            '/important_topic',
            String,
            self.callback,
            queue_size=10,
            tcp_nodelay=True
        )

    def publish_config(self, config):
        self.config_pub.publish(config)
        # Message persists even after publisher is destroyed
```

6. **Parameter Server Communication**:

```python
class ParameterHandler:
    def __init__(self):
        # Set parameters
        rospy.set_param('~max_velocity', 1.0)
        rospy.set_param('~min_distance', 0.5)
```

```
        # Get parameters
        self.max_vel = rospy.get_param('~max_velocity',
default=0.5)
        self.min_dist = rospy.get_param('~min_distance',
default=0.3)

        # Parameter update callback
        self.setup_param_callback()

    def setup_param_callback(self):
        rospy.Timer(rospy.Duration(1.0), self.check_params)

    def check_params(self, event):
        try:
            # Check for parameter updates
            new_max_vel = rospy.get_param('~max_velocity')
            if new_max_vel != self.max_vel:
                self.max_vel = new_max_vel
                self.handle_param_update()
        except:
            rospy.logwarn("Parameter not found")
```
```
```

Working with transforms (TF2) for robot coordination

1. **Basic TF2 Broadcasting**:

```python
#!/usr/bin/env python3
import rospy
import tf2_ros
import geometry_msgs.msg
from math import sin, cos

class TransformBroadcaster:
    def __init__(self):
        rospy.init_node('tf2_broadcaster')
        self.br = tf2_ros.TransformBroadcaster()
        self.rate = rospy.Rate(10)  # 10 Hz

    def broadcast_transform(self):
        while not rospy.is_shutdown():
            t = geometry_msgs.msg.TransformStamped()

            # Fill in transform data
            t.header.stamp = rospy.Time.now()
            t.header.frame_id = "base_link"
            t.child_frame_id = "laser"

            # Set translation
            t.transform.translation.x = 0.1
            t.transform.translation.y = 0.0
            t.transform.translation.z = 0.2

            # Set rotation (quaternion)
            t.transform.rotation.x = 0.0
            t.transform.rotation.y = 0.0
            t.transform.rotation.z = 0.0
```

```
            t.transform.rotation.w = 1.0

            # Broadcast transform
            self.br.sendTransform(t)
            self.rate.sleep()
```

2. **TF2 Listening and Frame Transformation**:

```python
class TransformListener:
    def __init__(self):
        self.tfBuffer = tf2_ros.Buffer()
        self.listener = tf2_ros.TransformListener(self.tfBuffer)

    def get_transform(self, target_frame, source_frame):
        try:
            # Look up transform
            trans = self.tfBuffer.lookup_transform(
                target_frame,
                source_frame,
                rospy.Time(0),
                rospy.Duration(1.0)
            )
            return trans
        except (tf2_ros.LookupException,
                tf2_ros.ConnectivityException,
                tf2_ros.ExtrapolationException) as e:
            rospy.logerr(f"Transform error: {e}")
            return None

    def transform_pose(self, pose, target_frame):
        try:
            # Transform pose from one frame to another
            pose_stamped = geometry_msgs.msg.PoseStamped()
```

```
            pose_stamped.pose = pose
            pose_stamped.header.frame_id = "base_link"
            pose_stamped.header.stamp = rospy.Time.now()

            transformed_pose = self.tfBuffer.transform(
                pose_stamped,
                target_frame,
                rospy.Duration(1.0)
            )
            return transformed_pose
        except tf2_ros.TransformException as e:
            rospy.logerr(f"Transform failed: {e}")
            return None
```

3. **Static Transform Publisher**:

```
```python
class StaticTransformPublisher:
    def __init__(self):
        self.static_broadcaster =
tf2_ros.StaticTransformBroadcaster()

        # Create static transform
        static_transformStamped =
geometry_msgs.msg.TransformStamped()

        static_transformStamped.header.stamp = rospy.Time.now()
        static_transformStamped.header.frame_id = "base_link"
        static_transformStamped.child_frame_id = "camera_link"

        static_transformStamped.transform.translation.x = 0.1
        static_transformStamped.transform.translation.y = 0.0
        static_transformStamped.transform.translation.z = 0.15
```
```

```
        static_transformStamped.transform.rotation.x = 0.0
        static_transformStamped.transform.rotation.y = 0.0
        static_transformStamped.transform.rotation.z = 0.0
        static_transformStamped.transform.rotation.w = 1.0


self.static_broadcaster.sendTransform(static_transformStamped)
```

4. **Dynamic Robot Transform Tree**:

```python
from tf.transformations import quaternion_from_euler

class RobotTransformTree:
    def __init__(self):
        self.br = tf2_ros.TransformBroadcaster()
        self.odom_sub = rospy.Subscriber('/odom', Odometry,
self.odom_callback)

    def odom_callback(self, msg):
        # Broadcast transform from odom to base_link
        t = geometry_msgs.msg.TransformStamped()

        t.header.stamp = msg.header.stamp
        t.header.frame_id = "odom"
        t.child_frame_id = "base_link"

        # Copy position
        t.transform.translation.x = msg.pose.pose.position.x
        t.transform.translation.y = msg.pose.pose.position.y
        t.transform.translation.z = msg.pose.pose.position.z
```

```
        # Copy orientation
        t.transform.rotation = msg.pose.pose.orientation

        self.br.sendTransform(t)
```

5. **Advanced TF2 Usage with Error Handling**:

```python
class AdvancedTransformHandler:
    def __init__(self):
        self.tfBuffer = tf2_ros.Buffer(rospy.Duration(30.0))  # 30 second buffer
        self.listener = tf2_ros.TransformListener(self.tfBuffer)

    def get_transform_with_timeout(self, target_frame, source_frame, timeout=1.0):
        start_time = rospy.Time.now()
        while not rospy.is_shutdown():
            try:
                transform = self.tfBuffer.lookup_transform(
                    target_frame,
                    source_frame,
                    rospy.Time(0)
                )
                return transform
            except tf2_ros.LookupException:
                if (rospy.Time.now() - start_time).to_sec() > timeout:
                    rospy.logerr("Transform lookup timed out")
                    return None
                rospy.sleep(0.1)
            except (tf2_ros.ConnectivityException,
```

```python
                tf2_ros.ExtrapolationException) as e:
            rospy.logerr(f"Transform error: {e}")
            return None

    def transform_point_cloud(self, cloud, target_frame):
        try:
            # Transform point cloud to target frame
            trans = self.tfBuffer.lookup_transform(
                target_frame,
                cloud.header.frame_id,
                cloud.header.stamp,
                rospy.Duration(1.0)
            )

            # Use tf2_sensor_msgs to transform point cloud
            transformed_cloud = do_transform_cloud(cloud, trans)
            return transformed_cloud

        except tf2_ros.TransformException as e:
            rospy.logerr(f"Point cloud transform failed: {e}")
            return None
```

How to use visualization tools like RViz effectively

1. **Basic RViz Configuration**:

````
```python
#!/usr/bin/env python3
import rospy
from visualization_msgs.msg import Marker, MarkerArray
from geometry_msgs.msg import Point

class RvizVisualizer:
    def __init__(self):
        rospy.init_node('rviz_visualizer')
        self.marker_pub =
rospy.Publisher('/visualization_marker', Marker, queue_size=10)
        self.marker_array_pub =
rospy.Publisher('/visualization_marker_array', MarkerArray,
queue_size=10)

    def create_basic_marker(self):
        marker = Marker()
        marker.header.frame_id = "map"
        marker.header.stamp = rospy.Time.now()
        marker.id = 0
        marker.type = Marker.SPHERE
        marker.action = Marker.ADD

        # Set position and scale
        marker.pose.position.x = 1.0
        marker.pose.position.y = 1.0
        marker.pose.position.z = 0.0
        marker.scale.x = 0.2
        marker.scale.y = 0.2
        marker.scale.z = 0.2

        # Set color (RGBA)
````
````

````
        marker.color.r = 1.0
        marker.color.g = 0.0
        marker.color.b = 0.0
        marker.color.a = 1.0

        return marker
```
````

2. **Advanced Visualization Techniques**:

````
```python
class AdvancedVisualizer:
    def __init__(self):
        self.path_pub = rospy.Publisher('/path_markers',
MarkerArray, queue_size=10)
        self.text_pub = rospy.Publisher('/text_markers',
MarkerArray, queue_size=10)

    def visualize_path(self, points):
        marker_array = MarkerArray()

        # Create line strip
        line_marker = Marker()
        line_marker.header.frame_id = "map"
        line_marker.type = Marker.LINE_STRIP
        line_marker.action = Marker.ADD
        line_marker.scale.x = 0.05  # line width

        # Add points to line strip
        for point in points:
            p = Point()
            p.x = point[0]
            p.y = point[1]
            p.z = 0.0
            line_marker.points.append(p)
````

```python
        marker_array.markers.append(line_marker)
        self.path_pub.publish(marker_array)

    def add_text_markers(self, positions, texts):
        marker_array = MarkerArray()

        for i, (pos, text) in enumerate(zip(positions, texts)):
            text_marker = Marker()
            text_marker.header.frame_id = "map"
            text_marker.type = Marker.TEXT_VIEW_FACING
            text_marker.id = i
            text_marker.text = text
            text_marker.pose.position.x = pos[0]
            text_marker.pose.position.y = pos[1]
            text_marker.scale.z = 0.3  # text size
            marker_array.markers.append(text_marker)

        self.text_pub.publish(marker_array)
```

3. **Interactive Markers**:

```python
from interactive_markers.interactive_marker_server import
InteractiveMarkerServer
from visualization_msgs.msg import InteractiveMarker,
InteractiveMarkerControl

class InteractiveVisualizer:
    def __init__(self):
        self.server = InteractiveMarkerServer("basic_controls")

    def make_interactive_marker(self):
        int_marker = InteractiveMarker()
```

```
        int_marker.header.frame_id = "base_link"
        int_marker.name = "my_marker"
        int_marker.description = "Simple 6-DOF Control"

        # Create 6-DOF control
        control = InteractiveMarkerControl()
        control.orientation.w = 1
        control.orientation.x = 1
        control.orientation.y = 0
        control.orientation.z = 0
        control.name = "rotate_x"
        control.interaction_mode =
InteractiveMarkerControl.ROTATE_AXIS
        int_marker.controls.append(control)

        self.server.insert(int_marker, self.process_feedback)
        self.server.applyChanges()

    def process_feedback(self, feedback):
        rospy.loginfo(f"Marker feedback:
{feedback.marker_name}")
```

4. **Custom RViz Displays**:

```python
from nav_msgs.msg import Path
from geometry_msgs.msg import PoseStamped

class CustomDisplays:
    def __init__(self):
        self.robot_path_pub = rospy.Publisher('/robot_path',
Path, queue_size=10)
```

```
        self.pose_array_pub = rospy.Publisher('/pose_array', 
PoseArray, queue_size=10)

    def publish_robot_path(self, poses):
        path = Path()
        path.header.frame_id = "map"
        path.header.stamp = rospy.Time.now()

        for pose in poses:
            pose_stamped = PoseStamped()
            pose_stamped.pose = pose
            path.poses.append(pose_stamped)

        self.robot_path_pub.publish(path)
```
```

5. **RViz Configuration Management**:

```
```python
class RvizConfigManager:
    def __init__(self):
        # Create RViz configuration file
        self.config = {
            'Visualization Manager': {
                'Global Options': {
                    'Fixed Frame': 'map',
                    'Background Color': '48; 48; 48'
                },
                'Displays': [
                    {
                        'Name': 'Grid',
                        'Class': 'rviz/Grid',
                        'Enabled': True,
                        'Cell Size': 1.0
                    },
```

```
                    {
                        'Name': 'Robot Model',
                        'Class': 'rviz/RobotModel',
                        'Enabled': True
                    },
                    {
                        'Name': 'LaserScan',
                        'Class': 'rviz/LaserScan',
                        'Enabled': True,
                        'Topic': '/scan'
                    }
                ]
            }
        }
```

6. **Real-time Data Visualization**:

```python
class RealTimeVisualizer:
    def __init__(self):
        self.pointcloud_pub =
rospy.Publisher('/visualize_pointcloud', PointCloud2,
queue_size=10)
        self.velocity_marker_pub =
rospy.Publisher('/velocity_markers', MarkerArray, queue_size=10)

    def visualize_velocity(self, linear_vel, angular_vel):
        marker_array = MarkerArray()

        # Create arrow for linear velocity
        linear_marker = Marker()
        linear_marker.type = Marker.ARROW
```

```
        linear_marker.scale.x = linear_vel
        linear_marker.scale.y = 0.1
        linear_marker.scale.z = 0.1
        linear_marker.color.r = 1.0
        linear_marker.color.a = 1.0

        # Create arc for angular velocity
        angular_marker = Marker()
        angular_marker.type = Marker.CYLINDER
        angular_marker.scale.x = abs(angular_vel)

        marker_array.markers = [linear_marker, angular_marker]
        self.velocity_marker_pub.publish(marker_array)
```