ROS2 Basics Challenge: Autonomous Delivery Robot

Challenge Overview

Create an Autonomous Delivery Robot system that demonstrates all ROS2 core concepts: Topics, Services, and Actions. This real-world scenario mimics actual robotics applications used in warehouses, hospitals, and factories.

Real-World Scenario

You're building a delivery robot that:

- Monitors sensors (Topics - continuous data)
- Accepts delivery requests (Services - immediate commands)
- Executes navigation missions (Actions - long-running tasks)
- Manages configuration (Parameters - runtime settings)

System Architecture

```text
DELIVERY ROBOT SYSTEM

┌─────────────────┬─────────────────┬─────────────────┐

│     TOPICS      │    SERVICES     │     ACTIONS     │

├─────────────────┼─────────────────┼─────────────────┤

│ Sensor Data     │ Quick Commands  │ Long Missions   │

│ Continuous      │ Immediate       │ Progress Updates│

│ One-to-Many     │ Request/Reply   │ Cancelable      │

│                 │                 │                 │
```

```
│ • Battery Level │ • Emergency Stop│ • Navigate to   │
│ • Lidar Scan    │ • System Check  │   Location      │
│ • Motor Odometry│ • Get Robot Info│ • Deliver Package│
│                 │                 │ • Charging Mission│
└─────────────────┴─────────────────┴─────────────────┘
```

Exercise Requirements

Part 1: Topics - Sensor Monitoring System

Create nodes that publish and subscribe to sensor data:

Nodes to implement:

- `battery_monitor`: Publishes battery level (0-100%)
- `sensor_processor`: Subscribes to battery and processes sensor fusion
- `safety_monitor`: Monitors all sensors for safety violations

Part 2: Services - Command System

Create request-response services for immediate commands:

Services to implement:

- `EmergencyStop` - Immediately stops the robot
- `SystemDiagnostic` - Returns robot health status
- `GetRobotInfo` - Returns current robot state

Part 3: Actions - Mission System

Create long-running tasks with progress feedback:

Actions to implement:

- `NavigateTo` - Move to specific coordinates with progress
- `DeliverPackage` - Complete delivery mission
- `AutoCharge` - Navigate to charging station

Part 4: Parameters - Configuration System

Make the robot configurable at runtime:

Parameters to implement:

- `max_speed`, `battery_low_threshold`, `safety_distance`
- `operation_mode`, `delivery_stations`, `emergency_contacts`

Exercise Requirements - Detailed Breakdown

📌 Part 1: Topics - Sensor Monitoring System

Node 1: battery_monitor.py

Purpose: Simulate battery monitoring and publish battery status

Features to implement:

- Publish BatteryStatus message every 2 seconds
- Simulate battery drain: -1% every 2 seconds when moving, +2% when charging
- Battery health states: "Good" (>70%), "Warning" (30-70%), "Critical" (<30%)
- Publish to topic: `/battery_level`

Message: BatteryStatus.msg

```text
float32 level     # 0-100%
```

```
string health_status   # "Good", "Warning", "Critical"

bool is_charging    # True/False

float32 voltage    # Simulated voltage
```

Expected Behavior:

- When robot is moving: Battery drains slowly
- When robot is charging: Battery increases
- Publishes continuous updates

Node 2: sensor_processor.py

Purpose: Subscribe to multiple sensors and fuse data

Subscriptions:

- `/battery_level` (BatteryStatus)
- `/lidar_scan` (LidarScan) - Simulated obstacle data
- `/motor_odometry` (Odometry) - Position data

Publications:

- `/sensor_fusion` (SensorFusion) - Combined sensor data
- *Note: Removed /safety_status - using /sensor_fusion for safety data*

Processing Logic:

- Combine battery + obstacle + position data
- Detect safety issues (low battery + obstacles)
- Calculate robot's safe operating envelope

Message Definitions:

LidarScan.msg

```
text
```

```
float32[] distances    # Distance readings at different angles
float32[] angles    # Corresponding angles
int32 obstacle_count  # Number of obstacles detected
```

Odometry.msg

```text
float32 x
float32 y
float32 theta    # Orientation
float32 linear_velocity
float32 angular_velocity
```

SensorFusion.msg

```text
float32 battery_level
int32 obstacle_count
float32 position_x
float32 position_y
string overall_status  # "SAFE", "CAUTION", "DANGER"
```

Node 3: safety_monitor.py

Purpose: Monitor all data streams for safety violations

Subscriptions:

- `/sensor_fusion` (SensorFusion)
- `/battery_level` (BatteryStatus)

Safety Rules:

- Battery < 20%: Warn and limit operations
- Obstacles < 0.5m: Emergency stop
- Multiple obstacles: Reduce speed
- System errors: Enter safe mode

Actions:

- Publish safety commands to `/emergency_cmd`
- Log safety events
- Trigger emergency procedures

Part 2: Services - Command System

Service 1: EmergencyStop.srv

Purpose: Immediate robot halt command

Service Definition:

```text
bool emergency    # True to activate emergency stop
---
bool success    # Whether stop was executed
string message    # Status message
```

Implementation in command_handler.py:

```python
def handle_emergency_stop(self, request, response):
    if request.emergency:
        # Immediate actions:
```

```python
        self.stop_all_motors()          # Implement this helper method
        self.cancel_all_actions()       # Implement this helper method
        self.publish_stop_signal()      # Implement this helper method
        response.success = True
        response.message = "Emergency stop activated - All systems halted"
    else:
        response.success = False
        response.message = "No emergency condition specified"
    return response
```

Service 2: SystemDiagnostic.srv

Purpose: Get comprehensive system health report

Service Definition:

```text
bool full_report    # True for detailed report
---
bool system_ok
string status_message
string[] components   # ["battery", "motors", "sensors", "navigation"]
string[] statuses    # ["OK", "WARNING", "ERROR"] corresponding to components
float32 battery_level
int32 error_count
```

Implementation Note:

```python
# Ensure arrays have equal length
assert len(components) == len(statuses)
```

Service 3: GetRobotInfo.srv

Purpose: Get current robot state information

Service Definition:

```text
---
string robot_state   # "IDLE", "NAVIGATING", "DELIVERING", "CHARGING", "ERROR", "RETURNING_TO_CHARGE"
float32 position_x
float32 position_y
float32 battery_level
string current_mission # Current mission description
string[] active_topics # List of active topics
```

Part 3: Actions - Mission System

Action 1: NavigateTo.action

Purpose: Long-running navigation with progress updates

Action Definition:

```text
# Goal
float32 target_x
float32 target_y
string location_name
float32 max_speed    # Optional speed limit
---
# Result
bool success
float32 final_x
float32 final_y
string message
float32 total_distance
float32 total_time
int32 obstacles_avoided
---
# Feedback
float32 current_x
float32 current_y
float32 progress_percentage
float32 distance_remaining
string status_message
int32 estimated_time_remaining
```

Corrected Implementation in navigation_controller.py:

```python
```

```python
async def execute_navigate_to(self, goal_handle):
    goal = goal_handle.request

    # 1. Validate goal
    if not self.is_valid_location(goal.target_x, goal.target_y):
        goal_handle.abort()
        return NavigateTo.Result(success=False, message="Invalid target
location")

    # Goal accepted - don't call succeed() until completion
    total_distance = self.calculate_distance(goal.target_x, goal.target_y)
    current_distance = 0
    start_x, start_y = self.current_position

    while current_distance < total_distance:
        # Check for cancellation
        if goal_handle.is_cancel_requested:
            result = NavigateTo.Result(
                success=False,
                message="Navigation cancelled",
                final_x=self.current_x,
                final_y=self.current_y
            )
            goal_handle.canceled()
            return result
```

```python
            # Update position and progress
            current_distance += self.move_step()
            progress = (current_distance / total_distance) * 100

            # Publish feedback
            feedback = NavigateTo.Feedback()
            feedback.current_x = self.current_x
            feedback.current_y = self.current_y
            feedback.progress_percentage = progress
            feedback.distance_remaining = total_distance - current_distance
            feedback.status_message = self.get_navigation_status()
            feedback.estimated_time_remaining =
self.calculate_eta(current_distance, total_distance)
            goal_handle.publish_feedback(feedback)

            await asyncio.sleep(0.1) # Simulate processing

        # 3. Return result - NOW call succeed()
        result = NavigateTo.Result(
            success=True,
            message=f"Arrived at {goal.location_name}",
            final_x=self.current_x,
            final_y=self.current_y,
            total_distance=total_distance,
            total_time=time.time() - self.mission_start_time,
            obstacles_avoided=self.obstacles_avoided
```

```python
    )
    goal_handle.succeed()
    return result
```

Action 2: DeliverPackage.action

Purpose: Complete delivery mission (navigation + package handling)

Action Definition:

```text
# Goal
string package_id
string destination_name
float32 destination_x
float32 destination_y
int32 priority    # 1-5, 5 is highest
---
# Result
bool success
string package_id
string delivery_status
float32 total_time
string message
---
# Feedback
string current_phase # "PLANNING", "NAVIGATING", "DELIVERING", "CONFIRMING"
```

```
float32 progress_percentage
string status_message
float32 time_elapsed
```

Action 3: AutoCharge.action

Purpose: Autonomous charging mission

Action Definition:

```text
# Goal
bool urgent    # True for low battery emergency
---
# Result
bool success
float32 battery_before
float32 battery_after
float32 charge_time
string message
---
# Feedback
string phase    # "NAVIGATING", "DOCKING", "CHARGING", "VERIFYING"
float32 battery_level
float32 progress_percentage
int32 time_remaining
```

Special Behavior:

- If `urgent=True`: Skip NAVIGATING phase and go directly to DOCKING
- Emergency charging protocol for critical battery levels

📌 Part 4: Parameters - Configuration System

Parameters in delivery_robot.py:

```python
from rcl_interfaces.msg import SetParametersResult

from rclpy.parameter import Parameter


def declare_parameters(self):

    # Navigation parameters

    self.declare_parameter('max_speed', 1.5)     # m/s

    self.declare_parameter('min_speed', 0.1)     # m/s

    self.declare_parameter('acceleration', 0.5)     # m/s²


    # Battery parameters

    self.declare_parameter('battery_low_threshold', 20.0)     # %

    self.declare_parameter('battery_critical_threshold', 10.0)   # %

    self.declare_parameter('charging_threshold', 30.0)     # %


    # Safety parameters

    self.declare_parameter('safety_distance', 0.5)     # meters

    self.declare_parameter('emergency_stop_distance', 0.2)     # meters
```

```
    self.declare_parameter('max_obstacle_count', 3)    # number

    # Operation parameters
    self.declare_parameter('operation_mode', 'normal')    # normal, cautious, aggressive
    self.declare_parameter('auto_charge', True)    # enable auto charging

    # Delivery stations (as string array)
    self.declare_parameter('delivery_stations', ['Reception', 'Office', 'Kitchen', 'Lab'])

    # Emergency contacts
    self.declare_parameter('emergency_contacts', ['operator1', 'admin'])
```

Corrected Parameter Validation:

```python
def parameter_callback(self, params):
    """Validate parameter changes"""
    for param in params:
        if param.name == 'max_speed' and param.value > 3.0:
            self.get_logger().error("Max speed cannot exceed 3.0 m/s for safety")
            return SetParametersResult(successful=False)

        if param.name == 'battery_low_threshold' and param.value < 5.0:
            self.get_logger().error("Battery low threshold too low - safety risk")
            return SetParametersResult(successful=False)
```

```
    return SetParametersResult(successful=True)
```

Corrected Parameter Setting in Error Handler:

```python
def handle_sensor_failure(self, sensor_type):
    """Handle sensor failures gracefully"""
    if sensor_type == "battery":
        self.get_logger().error("Battery sensor failure - using estimated values")
        self.estimated_battery_mode = True
    elif sensor_type == "lidar":
        self.get_logger().error("LIDAR failure - entering cautious mode")
        # Correct way to set parameters
        cautious_param = Parameter('operation_mode', value='cautious')
        self.set_parameters([cautious_param])
```

Integration Requirements

Corrected Data Flow Between Nodes:

```text
battery_monitor ─┐
                 ├─> sensor_processor ─> safety_monitor
lidar_simulator ─┘
                 ├─>delivery_robot ─>command_handler─>
                                                     navigation_controller
odometry_simulator ──────────────────────────────> mission_planner
```

Corrected State Management in delivery_robot.py:

```python
from enum import Enum


class RobotState(Enum):
    IDLE = "IDLE"
    NAVIGATING = "NAVIGATING"
    DELIVERING = "DELIVERING"
    CHARGING = "CHARGING"
    ERROR = "ERROR"
    EMERGENCY_STOP = "EMERGENCY_STOP"
    RETURNING_TO_CHARGE = "RETURNING_TO_CHARGE"  # Added missing state


def update_robot_state(self, new_state):
```

```
    old_state = self.current_state
    self.current_state = new_state

    self.get_logger().info(f"State change: {old_state} -> {new_state}")

    # State-specific actions
    if new_state == RobotState.EMERGENCY_STOP:
        self.execute_emergency_procedure()
    elif new_state == RobotState.CHARGING:
        self.start_charging_mission()
    elif new_state == RobotState.RETURNING_TO_CHARGE:
        self.initiate_charging_mission()
```

Parameter Change Broadcasting:

```python
def broadcast_parameter_update(self, param_name, param_value):
    """Notify all nodes of parameter changes"""
    # Publish to /config_update topic
    msg = ParameterUpdate()
    msg.parameter_name = param_name
    msg.parameter_value = str(param_value)
    self.config_publisher.publish(msg)
```

Expected Outputs by Node

Node 1: battery_monitor (Topic Publisher)

Purpose: Simulates battery monitoring system

Expected Output:

```text
[INFO] [1700000000.000] [battery_monitor]: Battery Monitor started

[INFO] [1700000002.000] [battery_monitor]: Battery Level: 95% | Health: Good

[INFO] [1700000004.000] [battery_monitor]: Battery Level: 94% | Health: Good

[INFO] [1700000006.000] [battery_monitor]: Battery Level: 92% | Health: Good

[INFO] [1700000008.000] [battery_monitor]: Battery Level: 85% | Health: Warning

[INFO] [1700000010.000] [battery_monitor]: Battery Level: 75% | Health: Critical

[INFO] [1700000012.000] [battery_monitor]: Battery Level: 100% | Health: Good (Charged)
```

Node 2: sensor_processor (Topic Subscriber/Publisher)

Purpose: Processes multiple sensor inputs and fuses data

Expected Output:

```text
[INFO] [1700000000.000] [sensor_processor]: Sensor Processor started
```

```
[INFO] [1700000000.100] [sensor_processor]: Subscribed to: /battery_level, /lidar_scan, /motor_odometry

[INFO] [1700000000.200] [sensor_processor]: Publishing to: /sensor_fusion

[INFO] [1700000001.000] [sensor_processor]: Sensor Fusion: All sensors OK | Obstacles: 0 | Position: (0.0, 0.0)

[INFO] [1700000002.000] [sensor_processor]: Sensor Fusion: Battery 95% | Obstacles: 1 | Position: (0.1, 0.0)

[INFO] [1700000003.000] [sensor_processor]: Safety Alert: Obstacle detected at 2.1m

[INFO] [1700000004.000] [sensor_processor]: Sensor Fusion: Battery 94% | Obstacles: 0 | Position: (0.3, 0.1)
```

Node 3: safety_monitor (Topic Subscriber)

Purpose: Monitors all sensors for safety violations

Expected Output:

```text
[INFO] [1700000000.000] [safety_monitor]: Safety Monitor started

[INFO] [1700000000.500] [safety_monitor]: All systems nominal

[INFO] [1700000002.500] [safety_monitor]: Monitoring: Battery, Obstacles, System Health

[INFO] [1700000005.000] [safety_monitor]: Warning: Battery below 30% (25%)

[INFO] [1700000007.000] [safety_monitor]: CRITICAL: Multiple obstacles detected! Auto-evasion engaged

[INFO] [1700000020.000] [safety_monitor]: Safety status cleared after auto-charge mission
```

Node 4: command_handler (Service Server)

Purpose: Handles immediate commands via services

Expected Output:

```text
[INFO] [1700000000.000] [command_handler]: Command Handler started
[INFO] [1700000000.100] [command_handler]: Services ready: /emergency_stop, /system_diagnostic, /get_robot_info
[INFO] [1700000010.000] [command_handler]: Received: SystemDiagnostic request
[INFO] [1700000010.001] [command_handler]: Diagnostic: Battery=85%, Motors=OK, Sensors=OK, Navigation=READY
[INFO] [1700000015.000] [command_handler]: EMERGENCY STOP ACTIVATED!
[INFO] [1700000015.001] [command_handler]: All systems halted safely
[INFO] [1700000020.000] [command_handler]: Received: GetRobotInfo request
[INFO] [1700000020.001] [command_handler]: Robot Info: State=IDLE, Position=(2.1, 3.4), Battery=82%
```

Node 5: navigation_controller (Action Server)

Purpose: Handles long-running navigation missions

Expected Output:

```text
[INFO] [1700000000.000] [navigation_controller]: Navigation Controller started
[INFO] [1700000000.100] [navigation_controller]: Action servers ready: /navigate_to, /deliver_package
```

```
[INFO] [1700000010.000] [navigation_controller]: Received navigation goal: Reception (5.0, 3.0)

[INFO] [1700000010.100] [navigation_controller]: Starting navigation to Reception...

[INFO] [1700000011.000] [navigation_controller]: Navigation: 10% complete | Position: (0.5, 0.3) | ETA: 9s

[INFO] [1700000012.000] [navigation_controller]: Navigation: 25% complete | Position: (1.2, 0.8) | ETA: 7s

[INFO] [1700000013.000] [navigation_controller]: Avoiding obstacle... recalculating route

[INFO] [1700000015.000] [navigation_controller]: Navigation: 60% complete | Position: (3.0, 1.8) | ETA: 4s

[INFO] [1700000018.000] [navigation_controller]: Navigation: 90% complete | Position: (4.5, 2.7) | ETA: 1s

[INFO] [1700000019.000] [navigation_controller]: Navigation completed:

Arrived at Reception
```

Node 6: mission_planner (Action Client)

Purpose: Plans and executes delivery missions

Expected Output:

```
text
```

```
[INFO] [1700000000.000] [mission_planner]: Mission Planner started

[INFO] [1700000005.000] [mission_planner]: New delivery mission: Package#123 - Office (8.0, 6.0)

[INFO] [1700000005.100] [mission_planner]: Sending navigation goal to Office...

[INFO] [1700000005.200] [mission_planner]: Goal accepted! Starting delivery mission
```

```
[INFO] [1700000006.000] [mission_planner]: Mission Progress: 15% | Moving to destination

[INFO] [1700000008.000] [mission_planner]: Mission Progress: 45% | Halfway to Office

[INFO] [1700000010.000] [mission_planner]: Mission Progress: 80% | Approaching target

[INFO] [1700000011.000] [mission_planner]: Arrived at Office! Delivering package...

[INFO] [1700000012.000] [mission_planner]: Delivery completed! Package#123 delivered successfully

[INFO] [1700000012.100] [mission_planner]: Mission Summary: Time=7s,

Distance=10.2m, Battery used=8%
```

Node 7: delivery_robot (Main Integration Node)

Purpose: Integrates all systems and manages robot state

Expected Output:

```text
[INFO] [1700000000.000] [delivery_robot]: Delivery Robot System ONLINE

[INFO] [1700000000.100] [delivery_robot]: Loading parameters: max_speed=1.5, battery_threshold=20

[INFO] [1700000000.200] [delivery_robot]: Subscribing to sensor topics...

[INFO] [1700000000.300] [delivery_robot]: Connecting to command services...

[INFO] [1700000000.400] [delivery_robot]: Connecting to action servers...

[INFO] [1700000000.500] [delivery_robot]: All systems connected and ready!

[INFO] [1700000005.000] [delivery_robot]: Robot State: IDLE | Battery: 95% | Position: (0.0, 0.0)
```

```
[INFO] [1700000010.000] [delivery_robot]: Received delivery request: Office (8.0, 6.0)

[INFO] [1700000010.100] [delivery_robot]: Robot State: NAVIGATING | Mission: Deliver to Office

[INFO] [1700000015.000] [delivery_robot]: Mission Update: 50% complete | Battery: 87%

[INFO] [1700000020.000] [delivery_robot]: Mission completed: Package delivered to Office

[INFO] [1700000020.100] [delivery_robot]: Robot State: IDLE | Position: (8.0, 6.0)

[INFO] [1700000025.000] [delivery_robot]: Battery low (22%) - Planning charging mission...
```

User Interaction Outputs

When User Runs Services:

```bash
# Terminal: User calls emergency stop

$ ros2 service call /emergency_stop delivery_interfaces/srv/EmergencyStop "{emergency: true}"


# Expected output in command_handler:

[INFO] [1700000030.000] [command_handler]: EMERGENCY STOP ACTIVATED by user!

[INFO] [1700000030.001] [command_handler]: Stopping all motors...

[INFO] [1700000030.002] [command_handler]: Cancelling all active missions...
```

```
[INFO] [1700000030.003] [command_handler]: Emergency stop complete - Robot SAFE
```

When User Sends Action Goals:

```bash
# Terminal: User starts navigation
$ ros2 action send_goal /navigate_to delivery_interfaces/action/NavigateTo "{
    target_x: 5.0,
    target_y: 3.0,
    location_name: 'Kitchen'
}" --feedback


# Expected output in mission_planner:
[INFO] [1700000040.000] [mission_planner]: User requested navigation to Kitchen
[INFO] [1700000040.100] [mission_planner]: Feedback: 0% | Calculating route...
[INFO] [1700000041.000] [mission_planner]: Feedback: 10% | Moving through corridor...
[INFO] [1700000042.000] [mission_planner]: Feedback: 35% | Avoiding table...
[INFO] [1700000043.000] [mission_planner]: Feedback: 70% | Approaching Kitchen...
[INFO] [1700000044.000] [mission_planner]: Result: Arrived at Kitchen successfully!
```

When User Changes Parameters:

```bash
```

```
# Terminal: User adjusts robot speed
$ ros2 param set /delivery_robot max_speed 2.0


# Expected output in delivery_robot:
[INFO] [1700000050.000] [delivery_robot]: Parameter updated: max_speed = 2.0
[INFO] [1700000050.001] [delivery_robot]: Broadcasting new speed limit to all systems...
[INFO] [1700000050.002] [navigation_controller]: Speed limit updated to 2.0 m/s
```

Error Scenario Outputs

Low Battery Scenario:

```text
[INFO] [1700000060.000] [battery_monitor]: Battery Level: 18% | Health: CRITICAL
[INFO] [1700000060.001] [safety_monitor]: CRITICAL: Battery below safety threshold (18%)
[INFO] [1700000060.002] [delivery_robot]: Low battery! Cancelling current mission...
[INFO] [1700000060.003] [mission_planner]: Mission cancelled: Low battery (18%)
[INFO] [1700000060.004] [delivery_robot]: Robot State: RETURNING_TO_CHARGE
```

Obstacle Detection Scenario:

```text
```

```
[INFO] [1700000070.000] [sensor_processor]: Obstacle detected at 1.5m - 30 degrees

[INFO] [1700000070.001] [safety_monitor]: Monitoring obstacle: Distance=1.5m, Bearing=30°

[INFO] [1700000070.002] [navigation_controller]: Recalculating route: Obstacle avoidance

[INFO] [1700000070.003] [mission_planner]: Feedback: Detour added - ETA increased by 5s
```

System Health Output

Startup Sequence:

```text
[INFO] [1700000000.000] [delivery_robot]: SYSTEM BOOT SEQUENCE STARTED

[INFO] [1700000000.100] [battery_monitor]: Battery system ONLINE

[INFO] [1700000000.200] [sensor_processor]: Sensor fusion ONLINE

[INFO] [1700000000.300] [safety_monitor]: Safety systems ARMED

[INFO] [1700000000.400] [command_handler]: Command services READY

[INFO] [1700000000.500] [navigation_controller]: Navigation systems CALIBRATED

[INFO] [1700000000.600] [mission_planner]: Mission planner INITIALIZED

[INFO] [1700000000.700] [delivery_robot]: ALL SYSTEMS GO! Delivery Robot READY
```

Detailed Implementation Tasks

Task 1: Create Interface Packages

```bash
# Create interfaces for all messages, services, actions
ros2 pkg create delivery_interfaces --build-type ament_cmake
```

Define in delivery_interfaces:

- `msg/`: BatteryStatus.msg, LidarScan.msg, Odometry.msg, SensorFusion.msg
- `srv/`: EmergencyStop.srv, SystemDiagnostic.srv, GetRobotInfo.srv
- `action/`: NavigateTo.action, DeliverPackage.action, AutoCharge.action

Task 2: Implement Topic-Based Nodes

File: battery_monitor.py

- Publishes battery level every 2 seconds
- Simulates battery drain during movement
- Publishes to: `/battery_level`

File: sensor_processor.py

- Subscribes to multiple sensor topics
- Performs sensor fusion
- Publishes processed data to: `/sensor_fusion`

Task 3: Implement Service-Based Nodes

File: command_handler.py

- Provides EmergencyStop service
- Provides SystemDiagnostic service
- Can stop robot immediately via service call

Task 4: Implement Action-Based Nodes

File: navigation_controller.py

- Action server for NavigateTo
- Provides progress feedback during navigation
- Handles cancellation correctly

Task 5: Implement Main Robot Brain

File: delivery_robot.py

- Uses all communication methods
- Subscribes to sensor topics
- Provides services for commands
- Calls actions for missions
- Manages parameters for configuration

Expected User Interaction

Using Topics (Monitoring):

```bash
# Monitor battery level

ros2 topic echo /battery_level


# Watch sensor fusion

ros2 topic echo /sensor_fusion
```

Using Services (Commands):

```bash
```

```bash
# Emergency stop
ros2 service call /emergency_stop delivery_interfaces/srv/EmergencyStop
"{emergency: true}"

# Get system status
ros2 service call /system_diagnostic
delivery_interfaces/srv/SystemDiagnostic
```

Using Actions (Missions):

```bash
# Start delivery mission
ros2 action send_goal /navigate_to delivery_interfaces/action/NavigateTo
"{target_x: 5.0, target_y: 3.0, location_name: 'Reception'}" --feedback

# Cancel if needed
ros2 action cancel_goal /navigate_to
```

Using Parameters (Configuration):

```bash
# Adjust robot settings
ros2 param set /delivery_robot max_speed 2.0
ros2 param set /delivery_robot battery_low_threshold 20

# Save configuration
ros2 param dump /delivery_robot
```

Starter Code Structure

```text
delivery_robot_ws/
    ├── delivery_interfaces/    # All message definitions
    │    ├── msg/
    │    ├── srv/
    │    ├── action/
    ├── sensor_nodes/    # Topic-based nodes
    │    ├── battery_monitor.py
    │    ├── sensor_processor.py
    │    ├── safety_monitor.py
    ├── command_nodes/    # Service-based nodes
    │    ├── command_handler.py
    ├── mission_nodes/    # Action-based nodes
    │    ├── navigation_controller.py
    │    ├── mission_planner.py
    ├── integration_nodes/    # Main system nodes
    └── delivery_robot.py
```

IMPORTANT NOTE: Data Simulation Strategy

Simulation Approach for This Exercise

Since this is a learning exercise without real hardware, ALL nodes will generate FAKE/SIMULATED data to demonstrate ROS2 concepts.

Data Simulation by Node

Node 1: battery_monitor.py

```python
# SIMULATED DATA: Battery levels
def simulate_battery(self):
    # Fake battery behavior:
    if self.robot_moving:
        self.battery_level -= 1.0  # Drain 1% every 2 seconds when moving
    elif self.robot_charging:
        self.battery_level += 2.0  # Charge when docked
    else:
        self.battery_level -= 0.1  # Slow drain when idle

    # Keep within bounds
    self.battery_level = max(0, min(100, self.battery_level))
```

Node 2: sensor_processor.py

```python
# SIMULATED DATA: Sensor readings
def simulate_lidar_data(self):
    # Fake obstacle detection
    import random
    obstacles = random.randint(0, 3)  # 0-3 random obstacles
    distances = [random.uniform(1.0, 5.0) for _ in range(obstacles)]
    return obstacles, distances
```

```python
def simulate_odometry(self):
    # Fake position updates
    self.current_x += random.uniform(-0.1, 0.1)
    self.current_y += random.uniform(-0.1, 0.1)
```

Node 3: safety_monitor.py

```python
# SIMULATED DATA: Safety events
def simulate_safety_events(self):
    # Random safety events for demonstration
    events = ["OBSTACLE_DETECTED", "LOW_BATTERY", "MOTOR_OVERHEAT"]
    return random.choice(events) if random.random() < 0.1 else None
```

Node 4: command_handler.py

```python
# SIMULATED DATA: System diagnostics
def simulate_diagnostic_data(self):
    return {
        'battery': random.randint(20, 100),
        'motors': random.choice(['OK', 'WARNING', 'ERROR']),
        'sensors': random.choice(['OK', 'CALIBRATING', 'ERROR']),
        'temperature': random.uniform(20.0, 45.0)
    }
```

Node 5: navigation_controller.py

```python
# SIMULATED DATA: Navigation progress
```

```
def simulate_navigation(self, target_x, target_y):
    # Fake movement toward target
    step_size = 0.1
    current_x, current_y = 0, 0  # Start from origin

    while self.distance_to_target(current_x, current_y, target_x, target_y)
> 0.1:
        # Move toward target
        dx = target_x - current_x
        dy = target_y - current_y
        distance = math.sqrt(dx**2 + dy**2)

        if distance > 0:
            current_x += (dx / distance) * step_size
            current_y += (dy / distance) * step_size

        yield current_x, current_y  # Yield progress
```

Node 6: mission_planner.py

```python
# SIMULATED DATA: Mission execution
def simulate_mission_timeline(self):
    # Fake mission phases and durations (consistent with 7-second total
mission)
    mission_phases = {
        'PLANNING': 1,
        'NAVIGATING': 5,
```

```
        'DELIVERING': 1,
        'CONFIRMING': 0
    }
    return mission_phases
```

Node 7: delivery_robot.py

```python
# SIMULATED DATA: Robot state and environment
def simulate_environment(self):
    # Fake environmental factors
    self.simulated_obstacles = random.randint(0, 2)
    self.simulated_battery_drain = random.uniform(0.05, 0.2)
    self.simulated_traffic_delay = random.uniform(0, 2.0)  # Reduced for consistency
```

Why Simulate Data?

Learning Benefits:

1. Focus on ROS2 Concepts: No hardware dependencies
2. Consistent Behavior: Predictable outputs for learning
3. Error Simulation: Can simulate various failure scenarios
4. Rapid Testing: No need for physical robot setup

Real Data vs Simulated Data:

Aspect	Real System	This Exercise
Battery	Actual voltage readings	Math simulation
Position	GPS/Odometry sensors	Coordinate calculation
Obstacles	LIDAR/Camera data	Random number generation
Commands	Motor controllers	Print statements

Simulation Realism:

- Battery drains faster when "moving"
- Navigation follows straight-line paths
- Obstacles appear randomly but logically
- System errors occur with low probability

Important for Learners:

- Don't worry about "fake" data - focus on ROS2 patterns

- All concepts transfer to real hardware later
- Simulation allows testing edge cases easily
- You're learning communication patterns, not sensor tech

The value is in understanding HOW to structure ROS2 nodes and communication, not in the data source!