Understanding ROS 2 Actions: The Robot Navigation Example

In ROS 2, actions handle long-running tasks that need progress updates and can be cancelled. While services are for quick requests, actions are for complex operations that take time. Think of actions as asynchronous tasks with feedback - like starting a download that shows progress percentage.

The Robot Navigation Analogy:

Imagine you're giving a delivery robot a mission 🚚:

- You (Action Client): "Navigate to position X=5, Y=3" → Action Goal
- Robot (Action Server): Starts moving and sends updates:
    - "25% complete, moving through corridor" → Feedback
    - "50% complete, avoiding obstacle" → Feedback
    - "100% complete, arrived at destination!" → Result
- You: Can cancel at any time: "Stop! Change destination!" → Cancellation

What We're Building:

In this exercise, we'll create a Navigation Action with two nodes:

- `navigation_action_server`: The "robot brain" that executes navigation missions
- `navigation_action_client`: The "mission control" that sends goals and monitors progress

Key Action Characteristics:

- Asynchronous: Client doesn't block while waiting
- Progress Updates: Real-time feedback during execution
- Cancellable: Can stop the task anytime
- Long-running: Perfect for complex operations
- Three-part: Goal, Feedback, Result

This pattern is essential for robot navigation, manipulation tasks, downloads, and any operation where you need to monitor progress and potentially cancel.

Now let's build our navigation action step by step! 🟧

Step 1: Create Action Package

bash

```bash
# Navigate to workspace src
cd ~/ros2_ws/src

# Create package
ros2 pkg create navigation_action --build-type ament_python
```

Step 2: Create Custom Action Interface

First, create the interface package for our custom action or just add to the existing interface that we use in the previous example my_interface:

bash

File: `my_interfaces/action/NavigateToPosition.action`

plaintext

```
# Goal Definition - Where we want the robot to navigate
float32 target_x          # Target X coordinate in meters
float32 target_y          # Target Y coordinate in meters
string location_name      # Human-readable destination name
int32 max_speed          # Maximum speed (optional parameter)
---
# Result Definition - What was achieved at the end
bool success              # Whether navigation was successful
float32 final_x          # Actual final X position
float32 final_y          # Actual final Y position
string message           # Result description
int32 total_time         # Total time taken in seconds
float32 distance_traveled # Total distance traveled in meters
---
# Feedback Definition - Progress updates during navigation
float32 current_x         # Current X position
float32 current_y         # Current Y position
```

```
float32 progress_percentage # Completion percentage (0-100)
string status_message     # Current status description
int32 estimated_time_remaining # Estimated seconds remaining
float32 current_speed     # Current speed in m/s
string current_zone       # Current area/zone the robot is in
```

Step 3: Configure Interface Package

File: `my_interfaces/CMakeLists.txt`

cmake

```
cmake_minimum_required(VERSION 3.8)

project(my_interfaces)



if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")

 add_compile_options(-Wall -Wextra -Wpedantic)

endif()



# Find dependencies

find_package(ament_cmake REQUIRED)

find_package(rosidl_default_generators REQUIRED)



# Generate interfaces

rosidl_generate_interfaces(${PROJECT_NAME}

 "srv/Calculator.srv"

 "action/NavigateToPosition.action"

)
```

```
# Export dependencies

ament_export_dependencies(rosidl_default_runtime)



# Install the srv directory

install(DIRECTORY srv

 DESTINATION share/${PROJECT_NAME}

)



ament_package()
```

File: `my_interfaces/package.xml`

```
xml
```

```
<?xml version="1.0"?>

<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypelocation="http://download.ros.org/schema/package_format3.xsd
package_format3.xsd"?>

<package format="3">

 <name>my_interfaces</name>

 <version>0.0.0</version>

 <description>Custom calculator service interfaces</description>

 <maintainer email="you@example.com">Your Name</maintainer>

 <license>Apache License 2.0</license>
```

```
 <buildtool_depend>ament_cmake</buildtool_depend>

 <buildtool_depend>rosidl_default_generators</buildtool_depend>

  <exec_depend>rosidl_default_runtime</exec_depend>

 <member_of_group>rosidl_interface_packages</member_of_group>

</package>
```

Step 4: Create Action Server Code

File: `navigation_action/navigation_action/server.py`

python

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

import rclpy

from rclpy.action import ActionServer

from rclpy.node import Node

import time

import math

import random



from my_interfaces.action import NavigateToPosition



class NavigationActionServer(Node):

   def __init__(self):

       super().__init__('navigation_action_server')



       # Create action server
```

```
        self._action_server = ActionServer(
            self,
            NavigateToPosition,
            'navigate_to_position',
            self.execute_callback
        )

        self.get_logger().info('🚀 Navigation Action Server ready!')
        self.get_logger().info('🎯 Waiting for navigation goals...')

    def execute_callback(self, goal_handle):
        self.get_logger().info(f'🎯 Received navigation goal: {goal_handle.request.location_name} '
                               f'({goal_handle.request.target_x}, {goal_handle.request.target_y})')

        # Start executing the action
        goal_handle.succeed()

        # Initialize feedback message
        feedback_msg = NavigateToPosition.Feedback()

        # Simulate robot starting position
        current_x = 0.0
        current_y = 0.0
        target_x = goal_handle.request.target_x
```

```
        target_y = goal_handle.request.target_y

        # Calculate total distance
        total_distance = math.sqrt(target_x**2 + target_y**2)

        # Simulate navigation process (more realistic with 20 steps)
        step_size = total_distance / 20
        steps = 20

        for step in range(1, steps + 1):
            # Check if goal was cancelled
            if goal_handle.is_cancel_requested:
                goal_handle.canceled()
                self.get_logger().info('❌ Navigation cancelled by
user')
                result = NavigateToPosition.Result()
                result.success = False
                result.final_x = current_x
                result.final_y = current_y
                result.message = "Navigation cancelled by user"
                result.total_time = step  # FIX: This is now integer
                result.distance_traveled = math.sqrt(current_x**2 +
current_y**2)
                return result

            # Simulate movement with some randomness for realism
```

```
            progress = step / steps
            current_x = target_x * progress + random.uniform(-0.1, 0.1)
            current_y = target_y * progress + random.uniform(-0.1, 0.1)

            # Update feedback
            feedback_msg.current_x = current_x
            feedback_msg.current_y = current_y
            feedback_msg.progress_percentage = progress * 100.0
            feedback_msg.estimated_time_remaining = steps - step  # FIX: Integer
            feedback_msg.current_speed = random.uniform(0.5, 1.5)

            # Simulate different status messages based on progress
            if progress < 0.2:
                feedback_msg.status_message = "Starting motors, initializing navigation system..."
                feedback_msg.current_zone = "Start Area"
            elif progress < 0.4:
                feedback_msg.status_message = "Moving through main corridor..."
                feedback_msg.current_zone = "Main Corridor"
            elif progress < 0.6:
                feedback_msg.status_message = "Avoiding dynamic obstacles..."
                feedback_msg.current_zone = "Central Area"
            elif progress < 0.8:
```

```
                feedback_msg.status_message = "Approaching target
area..."

                feedback_msg.current_zone = "Destination Zone"

            else:

                feedback_msg.status_message = "Fine-tuning final
position..."

                feedback_msg.current_zone = "Target Area"



            # Publish feedback

            goal_handle.publish_feedback(feedback_msg)

            self.get_logger().info(

                f'📊 Progress: {progress*100:.1f}% | '

                f'Position: ({current_x:.2f}, {current_y:.2f}) | '

                f'Zone: {feedback_msg.current_zone}'

            )



            # Simulate processing time

            time.sleep(0.5)  # Faster updates for better user experience



        # Action completed successfully

        self.get_logger().info(f'✅ Navigation completed! Arrived at
{goal_handle.request.location_name}')



        # Set result - FIX ALL INTEGER FIELDS

        result = NavigateToPosition.Result()

        result.success = True
```

```
        result.final_x = target_x

        result.final_y = target_y

        result.message = f"Successfully arrived at
{goal_handle.request.location_name}"

        result.total_time = int(steps * 0.5)  # FIX: Convert to integer

        result.distance_traveled = total_distance



        return result



def main():

    rclpy.init()

    navigation_server = NavigationActionServer()



    try:

        rclpy.spin(navigation_server)

    except KeyboardInterrupt:

        print("\n🛑 Navigation server shutdown requested")

    finally:

        navigation_server.destroy_node()

        rclpy.shutdown()

        print("✅ Navigation server shutdown complete")



if __name__ == '__main__':

    main()
```

Step 5: Create Action Client Code

File: `navigation_action/navigation_action/client.py`

python

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

import rclpy

from rclpy.action import ActionClient

from rclpy.node import Node

import sys

import threading



from my_interfaces.action import NavigateToPosition  # CHANGE IMPORT



class NavigationActionClient(Node):

   def __init__(self):

       super().__init__('navigation_action_client')



       # Create action client

       self._action_client = ActionClient(

           self,

           NavigateToPosition,

           'navigate_to_position'

       )



       self.get_logger().info('🎮 Navigation Action Client ready!')
```

```
        # Track current goal
        self._goal_handle = None
        self._send_goal_future = None
        self._get_result_future = None

    def wait_for_server(self):
        self.get_logger().info('⏳ Waiting for action server...')
        return self._action_client.wait_for_server(timeout_sec=10.0)

    def send_goal(self, target_x, target_y, location_name, max_speed=1):
        # Create goal message
        goal_msg = NavigateToPosition.Goal()
        goal_msg.target_x = target_x
        goal_msg.target_y = target_y
        goal_msg.location_name = location_name
        goal_msg.max_speed = max_speed

        self.get_logger().info(f'🎯 Sending navigation goal:
{location_name} ({target_x}, {target_y})')

        # Send goal and set up callbacks
        self._send_goal_future = self._action_client.send_goal_async(
            goal_msg,
            feedback_callback=self.feedback_callback
        )
```

```
self._send_goal_future.add_done_callback(self.goal_response_callback)


    def goal_response_callback(self, future):
        self._goal_handle = future.result()

        if not self._goal_handle.accepted:
            self.get_logger().error('❌ Goal rejected by server')
            return

        self.get_logger().info('✅ Goal accepted by server,
navigating...')

        # Get result
        self._get_result_future = self._goal_handle.get_result_async()

self._get_result_future.add_done_callback(self.get_result_callback)


    def get_result_callback(self, future):
        result = future.result().result
        if result.success:
            self.get_logger().info(f'🏁 {result.message}')
            self.get_logger().info(f'📍 Final position:
({result.final_x:.2f}, {result.final_y:.2f})')
            self.get_logger().info(f'⏱️  Total time: {result.total_time}
seconds')
```

```
            self.get_logger().info(f'📏 Distance traveled:
{result.distance_traveled:.2f} meters')
        else:
            self.get_logger().error(f'💥 {result.message}')

        # Shutdown after receiving result
        self.destroy_node()
        rclpy.shutdown()

    def feedback_callback(self, feedback_msg):
        feedback = feedback_msg.feedback
        self.get_logger().info(
            f'📊 Progress: {feedback.progress_percentage:.1f}% | '
            f'Position: ({feedback.current_x:.2f},
{feedback.current_y:.2f}) | '
            f'Speed: {feedback.current_speed:.1f}m/s | '
            f'Zone: {feedback.current_zone} | '
            f'ETA: {feedback.estimated_time_remaining}s'
        )
        self.get_logger().info(f'    📝 Status:
{feedback.status_message}')

    def cancel_goal(self):
        if self._goal_handle:
            self.get_logger().info('🛑 Cancelling current
navigation...')
            future = self._goal_handle.cancel_goal_async()
```

```
            future.add_done_callback(self.cancel_done_callback)


    def cancel_done_callback(self, future):
        cancel_response = future.result()
        if len(cancel_response.goals_canceling) > 0:
            self.get_logger().info('✅ Navigation successfully cancelled')
        else:
            self.get_logger().info('❌ Failed to cancel navigation')


def main():
    rclpy.init()


    client = NavigationActionClient()


    # Wait for server
    if not client.wait_for_server():
        client.get_logger().error('❌ Action server not available after 10 seconds')
        client.destroy_node()
        rclpy.shutdown()
        return


    # Get command line arguments or use defaults
    if len(sys.argv) >= 4:
        target_x = float(sys.argv[1])
```

```
        target_y = float(sys.argv[2])
        location_name = sys.argv[3]
        max_speed = int(sys.argv[4]) if len(sys.argv) > 4 else 1
    else:
        # Default mission
        target_x = 5.0
        target_y = 3.0
        location_name = "Storage Room"
        max_speed = 1
        print("ℹ️  Using default mission: Storage Room (5.0, 3.0)")
        print("💡 Usage: ros2 run navigation_action client <x> <y> <location_name> [max_speed]")

    # Send goal
    client.send_goal(target_x, target_y, location_name, max_speed)

    # Start a thread to handle user input for cancellation
    def user_input_thread():
        while rclpy.ok():
            try:
                user_input = input("\nPress 'c' to cancel navigation, 'q' to quit: ").strip().lower()
                if user_input == 'c':
                    client.cancel_goal()
                elif user_input == 'q':
                    client.destroy_node()
```

```
                    rclpy.shutdown()
                    break
            except:
                break

    input_thread = threading.Thread(target=user_input_thread,
daemon=True)
    input_thread.start()

    # Spin in main thread
    try:
        rclpy.spin(client)
    except KeyboardInterrupt:
        print("\n🛑 Client shutdown requested")
    finally:
        if not client.destroyed:
            client.destroy_node()
        rclpy.shutdown()

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

Step 6: Configure Navigation Action Package

File: `navigation_action/package.xml`

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypelocation="http://download.ros.org/schema/package_format3.xsd
package_format3.xsd"?>
<package format="3">
  <name>navigation_action</name>
  <version>0.0.0</version>
  <description>ROS 2 Action example for robot navigation</description>
  <maintainer email="you@example.com">Your Name</maintainer>
  <license>Apache License 2.0</license>

  <depend>rclpy</depend>
  <depend>my_interfaces</depend>

  <buildtool_depend>ament_python</buildtool_depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>

</package>
```

File: `navigation_action/setup.py`

```python
from setuptools import setup

package_name = 'navigation_action'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
```

```
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='your_name',
    maintainer_email='your_email@example.com',
    description='ROS 2 Action example for robot navigation',
    license='Apache License 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'navigation_server = navigation_action.server:main',
            'navigation_client = navigation_action.client:main',
        ],
    },
)
```

Step 7: Build and Run

```
bash
# Navigate to workspace
cd ~/ros2_ws

# Build all packages
colcon build --packages-select my_interfaces navigation_action

# Source the workspace
source install/setup.bash
```

Step 8: Run the Navigation Example

Terminal 1 - Start Action Server:

```
bash
ros2 run navigation_action navigation_server
```

Output:

```
text
```

```
[INFO] [1700000000.000000000] [navigation_action_server]: 🚀 Navigation Action Server ready!
[INFO] [1700000000.000000000] [navigation_action_server]: 🎯 Waiting for navigation goals...
```

Terminal 2 - Run Action Client:

```
bash
```

```
ros2 run navigation_action navigation_client 8.0 6.0 "Kitchen"
```

Output:

```
text
```

```
[INFO] [1700000000.100000000] [navigation_action_client]: 🎮 Navigation Action Client ready!
[INFO] [1700000000.100000000] [navigation_action_client]: ⏳ Waiting for action server...
[INFO] [1700000000.200000000] [navigation_action_client]: 🎯 Sending navigation goal: Kitchen (8.0, 6.0)
[INFO] [1700000000.300000000] [navigation_action_client]: ✅ Goal accepted by server, navigating...
[INFO] [1700000000.400000000] [navigation_action_client]: 📊 Progress: 10.0% | Position: (0.8, 0.6) | ETA: 9s | Status: Starting motors, initializing navigation...
[INFO] [1700000001.400000000] [navigation_action_client]: 📊 Progress: 20.0% | Position: (1.6, 1.2) | ETA: 8s | Status: Moving through main corridor...
```

Terminal 3 - Monitor Actions:

```
bash
```

```
# List all actions
ros2 action list

# Get action info
ros2 action info /navigate_to_position

# Send goal directly from CLI
```

```
ros2 action send_goal /navigate_to_position
navigation_interfaces/action/NavigateToPosition "{target_x: 3.0, target_y:
4.0, location_name: 'Office'}"

# Send goal with feedback
ros2 action send_goal /navigate_to_position
navigation_interfaces/action/NavigateToPosition "{target_x: 2.0, target_y:
2.0, location_name: 'Lab'}" --feedback
```

Key Features Demonstrated:

1. 🎯 Goal Setting: Send navigation destinations
2. 📊 Progress Feedback: Real-time position and status updates
3. 🛑 Cancellation: Stop navigation anytime
4. 🏁 Result Handling: Success/failure outcomes
5. ⏱️ Time Estimation: ETA updates during execution

Real-World Applications:

- Robot Navigation: Move to specific coordinates
- Manipulation Tasks: Pick and place operations
- Data Processing: Long computations with progress
- Downloads: File transfers with progress percentage
- Mission Planning: Complex multi-step operations

This example shows why actions are perfect for any task that takes time, needs progress updates, and might need to be cancelled! 🚀

ROS 2 Action Principal Functions Reference Table

Action Server Functions

Function	Description	Usage Example	When to Use
`ActionServer()`	Creates an action server	`self._action_server = ActionServer(self, NavigateToPosition, 'navigate_to_position', execute_callback)`	Initialize action server in node constructor
`goal_handle.succeed()`	Marks goal as accepted and starts execution	`goal_handle.succeed()`	After validating goal, before starting execution
`goal_handle.publish_feedback()`	Sends progress updates to client	`goal_handle.publish_feedback(feedback_msg)`	During long-running tasks to show progress
`goal_handle.is_cancel_requested`	Checks if cancellation was requested	`if goal_handle.is_cancel_requested:`	Periodically during execution to handle cancellations
`goal_handle.canceled()`	Marks goal as cancelled	`goal_handle.canceled()`	When user requests cancellation
`return result`	Sends final result to client	`return result`	When action completes (success or failure)

Action Client Functions

Function	Description	Usage Example	When to Use
`ActionClient()`	Creates an action client	`self._action_client = ActionClient(self, NavigateToPosition, 'navigate_to_position')`	Initialize action client in node constructor
`wait_for_server()`	Waits for action server to be available	`self._action_client.wait_for_server(timeout_sec=10.0)`	Before sending goals, ensure server is ready
`send_goal_async()`	Sends goal to server asynchronously	`self._send_goal_future = self._action_client.send_goal_async(goal_msg, feedback_callback)`	To start an action without blocking
`goal_handle.get_result_async()`	Gets result asynchronously	`self._get_result_future = self._goal_handle.get_result_async()`	After goal is accepted, to get final result
`cancel_goal_async()`	Cancels current goal	`future = self._goal_handle.cancel_goal_async()`	When user wants to stop execution
`add_done_callback()`	Adds callback for async operations	`future.add_done_callback(self.goal_response_callback)`	To handle completion of async operations

Callback Functions Pattern

Callback Type	Purpose	Signature	Example Usage
Execute Callback	Server-side goal execution	`def execute_callback(self, goal_handle):`	Contains the main action logic
Goal Response Callback	Client-side goal acceptance	`def goal_response_callback(self, future):`	Handle whether goal was accepted/rejected
Result Callback	Client-side result handling	`def get_result_callback(self, future):`	Process the final result from server
Feedback Callback	Client-side progress updates	`def feedback_callback(self, feedback_msg):`	Update UI with progress information
Cancel Done Callback	Client-side cancellation confirmation	`def cancel_done_callback(self, future):`	