ROS 2 Service Implementation: Two Approaches Compared

Introduction

In ROS 2, while topics handle continuous data streams, services provide synchronous request-response communication - perfect for commands that need immediate results. Think of services as remote function calls between nodes: one node asks a question, another provides the answer.

The Calculator Analogy:

Imagine you're in a restaurant 🟩:

- You (Client): "What's 5 + 3?" → Service Request
- Waiter (Service): Takes your question to the kitchen
- Chef (Server): Calculates the answer
- Waiter: "It's 8!" → Service Response

Two Implementation Approaches

Approach 1: Custom Python Architecture (Publisher-Subscriber Pattern)

What We Built:

A custom request-response system using ROS 2 topics instead of true services.

Directory Structure:

```
text
simple_service/ (ament_python)
├── simple_service/
│   ├── calculator.py      # Custom Python classes
│   ├── server.py          # Listens on 'calculator_requests' topic
│   ├── client.py          # Publishes to 'calculator_requests'
│   └── cli_client.py      # Custom CLI tool
├── package.xml
└── setup.py
```

Key Components:

- Custom Python Classes: `CalculatorRequest`, `CalculatorResponse` defined in Python
- JSON Serialization: Manual message formatting
- Topic-based: Uses publisher-subscriber pattern
- Custom CLI: Required writing `cli_client.py`

Approach 2: True ROS 2 Services (Standard Approach)

What We Built:

A proper ROS 2 service using interface definition files and the standard service-client pattern.

Directory Structure:

```
text
ros2_ws/src/
├── my_interfaces/ (ament_cmake)          # Interface package
│   ├── srv/
│   │   └── Calculator.srv                # Service definition
│   ├── CMakeLists.txt
│   └── package.xml
└── simple_service/ (ament_python)        # Implementation package
    ├── simple_service/
    │   ├── true_server.py               # True service server
    │   └── true_client.py               # True service client
    ├── package.xml
    └── setup.py
```

Key Components:

- Service Definition: `Calculator.srv` file
- Auto-generated Code: ROS 2 generates Python classes automatically
- Standard Service Pattern: Uses ROS 2 service-client infrastructure
- Universal CLI: Works with built-in `ros2 service call`

Usage:

```
bash
# Terminal 1 - Start Server
ros2 run simple_service true_server

# Terminal 2 - Use Standard ROS 2 CLI (no custom code needed)
ros2 service call /calculator my_interfaces/srv/Calculator "{a: 10, b: 20, operation: 'add'}"
```

Key Service Characteristics Comparison

Characteristic	Custom Python Architecture	True ROS 2 Services
Synchronous	✅ Manual implementation	✅ Built-in
One-to-One	✅ Request-response pattern	✅ Native service pattern
CLI Support	❌ Requires custom client	✅ `ros2 service call`
Type Safety	❌ Manual validation	✅ Build-time checking
Multi-language	❌ Python only	✅ Python, C++, etc.
Service Discovery	❌ Manual topic management	✅ Automatic via ROS 2
Standard Practice	❌ Custom solution	✅ ROS 2 standard

When to Use Each Approach

Use Custom Python Architecture When:

- 🎓 Learning/experimentation - simpler to understand
- 🚀 Rapid prototyping - faster iteration
- 🐍 Python-only projects - no need for multi-language support
- 🎯 Full control needed - custom serialization/validation

Use True ROS 2 Services When:

- 🏭 Production systems - robust and standardized
- 🌐 Multi-language projects - C++, Python, etc.

- 🔧 Integration with ROS 2 tools - rqt, CLI, etc.
- 📊 Team development - following ROS 2 best practices
- 🔄 Future maintenance - standard patterns

Key Benefits Summary

Custom Python Architecture Benefits:

- ✅ Simplicity: Easier to understand for beginners
- ✅ Full Control: Complete control over serialization and logic
- ✅ No Separate Packages: Everything in one Python package
- ✅ Rapid Development: Quick to implement and modify

True ROS 2 Services Benefits:

- ✅ Standard Tools: Works with `ros2 service call`, `rqt`, etc.
- ✅ Type Safety: Compile-time type checking
- ✅ Multi-language: Same interfaces work for Python, C++, etc.
- ✅ Service Discovery: Automatic service registration and discovery
- ✅ Production Ready: Follows ROS 2 best practices
- ✅ Better Debugging: Standardized tools and patterns

Conclusion

Both approaches can solve the calculator service problem, but they represent different levels of ROS 2 integration:

- Custom Python Architecture: Great for learning and Python-only projects, but requires more manual work and doesn't integrate with standard ROS 2 tools.
- True ROS 2 Services: The standard approach that provides full integration with the ROS 2 ecosystem, type safety, and multi-language support.

For most ROS 2 projects, especially those intended for production or team development, True ROS 2 Services is the recommended approach as it leverages the full power of the ROS 2 framework and follows established best practices.

The journey from custom implementation to standardized solution demonstrates the importance of understanding ROS 2's architecture and choosing the right tool for your specific use case! 🚀

Step-by-Step Implementation

Step 1: Create Interface Package

```
cd ~/ros2_ws/src

ros2 pkg create my_interfaces --build-type ament_cmake
```

Step 2: Create Service Definition

File: `my_interfaces/srv/Calculator.srv`

```

int64 a

int64 b

string operation

---

int64 result

bool success

string message
```

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_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)


rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/Calculator.srv"
)


ament_export_dependencies(rosidl_default_runtime)


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: Build Interface Package First

```
bash
cd ~/ros2_ws
colcon build --packages-select my_interfaces
source install/setup.bash
```

Step 5: Create True Server in simple_service

File: `simple_service/simple_service/true_server.py`

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

```
from my_interfaces.srv import Calculator  # Import from your custom interface!


class TrueCalculatorServer(Node):
    def __init__(self):
        super().__init__('true_calculator_server')

        # Create a TRUE ROS 2 service
        self.srv = self.create_service(
            Calculator,
            'calculator',
            self.calculate_callback
        )

        self.get_logger().info('🎯 True Calculator Server ready!')
        self.get_logger().info('📡 Service: /calculator')
        self.get_logger().info('💡 Use: ros2 service call /calculator my_interfaces/srv/Calculator "{a: 10, b: 20, operation: \'add\'}"')

    def calculate_callback(self, request, response):
        self.get_logger().info(
            f'📥 Received: {request.a} {request.operation} {request.b}'
        )

        try:
            if request.operation == "add":
                response.result = request.a + request.b
```

```
                response.success = True
                response.message = "Addition successful"
            elif request.operation == "subtract":
                response.result = request.a - request.b
                response.success = True
                response.message = "Subtraction successful"
            elif request.operation == "multiply":
                response.result = request.a * request.b
                response.success = True
                response.message = "Multiplication successful"
            elif request.operation == "divide":
                if request.b == 0:
                    response.success = False
                    response.message = "Division by zero error"
                    response.result = 0
                else:
                    response.result = request.a // request.b
                    response.success = True
                    response.message = "Division successful"
            else:
                response.success = False
                response.message = f"Unknown operation:
{request.operation}"
                response.result = 0


        except Exception as e:
```

```
            response.success = False
            response.message = f"Calculation error: {str(e)}"
            response.result = 0

        self.get_logger().info(f'📤 Sending: {response.message} - Result: {response.result}')
        return response

def main():
    rclpy.init()
    server = TrueCalculatorServer()

    try:
        rclpy.spin(server)
    except KeyboardInterrupt:
        print("\n🛑 Server shutdown requested")
    finally:
        server.destroy_node()
        rclpy.shutdown()
        print("✅ Server shutdown complete")

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

Step 6: Create True Client in simple_service

File: `simple_service/simple_service/true_client.py`

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

import rclpy

from rclpy.node import Node

from my_interfaces.srv import Calculator

import sys


class TrueCalculatorClient(Node):

    def __init__(self):

        super().__init__('true_calculator_client')

        self.cli = self.create_client(Calculator, 'calculator')


        self.get_logger().info('⏳ Waiting for server...')

        while not self.cli.wait_for_service(timeout_sec=1.0):

            self.get_logger().info('⏳ Service not available, waiting...')


    def send_request(self, a, b, operation):

        request = Calculator.Request()

        request.a = a

        request.b = b

        request.operation = operation


        future = self.cli.call_async(request)

        return future


def main():
```

```
    rclpy.init()

    client = TrueCalculatorClient()


    # Get command line arguments or use defaults

    if len(sys.argv) == 4:

        a = int(sys.argv[1])

        b = int(sys.argv[2])

        operation = sys.argv[3]

    else:

        # Default values for testing

        a = 15

        b = 3

        operation = "add"

        print("ℹ️  Using default values: 15, 3, 'add'")

        print("💡 Usage: ros2 run simple_service true_client <a> <b>
<operation>")


    # Send request

    future = client.send_request(a, b, operation)

    client.get_logger().info(f'📤 Sent request: {a} {operation} {b}')


    # Wait for response

    client.get_logger().info('⏳ Waiting for response...')

    rclpy.spin_until_future_complete(client, future)


    if future.result() is not None:
```

```
        response = future.result()
        if response.success:
            client.get_logger().info(f'✅ {response.message}')
            client.get_logger().info(f'🎯 Result: {response.result}')
        else:
            client.get_logger().error(f'❌ {response.message}')
    else:
        client.get_logger().error('💥 Service call failed')

    client.destroy_node()
    rclpy.shutdown()

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

Step 7: Update simple_service package.xml

File: `simple_service/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>simple_service</name>
  <version>0.0.0</version>
  <description>Complete service example with both custom and true ROS 2
services</description>
```

```
  <maintainer email="you@example.com">Your Name</maintainer>
  <license>Apache License 2.0</license>

  <depend>rclpy</depend>
  <depend>std_msgs</depend>
  <depend>my_interfaces</depend>  <!-- ADD THIS DEPENDENCY -->

  <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>
```

Step 8: Update simple_service setup.py

File: `simple_service/setup.py`

```
python
from setuptools import setup

package_name = 'simple_service'
```

```
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='Complete service example with both custom and true ROS 2 services',
    license='Apache License 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            # Custom Python Architecture
            'server = simple_service.server:main',
            'client = simple_service.client:main',
            'calculator_call = simple_service.cli_client:main',

            # True ROS 2 Services
            'true_server = simple_service.true_server:main',
```

```
            'true_client = simple_service.true_client:main',
        ],
    },
)
```

Step 9: Build Everything

```
bash
cd ~/ros2_ws
colcon build --packages-select my_interfaces simple_service
source install/setup.bash
```

Usage Examples

Method 1: Using True ROS 2 Services (Recommended)

Terminal 1 - Start True Server:

```
bash
ros2 run simple_service true_server
```

Terminal 2 - Use ros2 service call (No custom code needed!):

```
bash
# List all services (you'll see /calculator!)
ros2 service list

# Get service info
ros2 service type /calculator

# Call the service directly!
```

```
ros2 service call /calculator my_interfaces/srv/Calculator "{a: 10, b: 20,
operation: 'add'}"



# Test other operations

ros2 service call /calculator my_interfaces/srv/Calculator "{a: 15, b: 3,
operation: 'multiply'}"

ros2 service call /calculator my_interfaces/srv/Calculator "{a: 10, b: 0,

operation: 'divide'}"
```

Terminal 3 - Use the True Client:

```
bash

ros2 run simple_service true_client 25 5 multiply
```