

CSC398 Introduction to Autonomous Robots

- Introduction to ROS 2 (4) -

Ubbo Visser

Department of Computer Science
College of Arts and Sciences
University of Miami

September 2026



- ▶ ROS 2 services
- ▶ ROS 2 actions
- ▶ ROS 2 time
- ▶ ROS 2 bags (rosbag2)
- ▶ Debugging strategies



ROS 2 SERVICES

- ▶ Request/response communication between nodes is implemented using **services**
- ▶ The *service server* provides the service
- ▶ The *service client* sends a request to that service
- ▶ Services are defined using **.srv interface files**

List services:

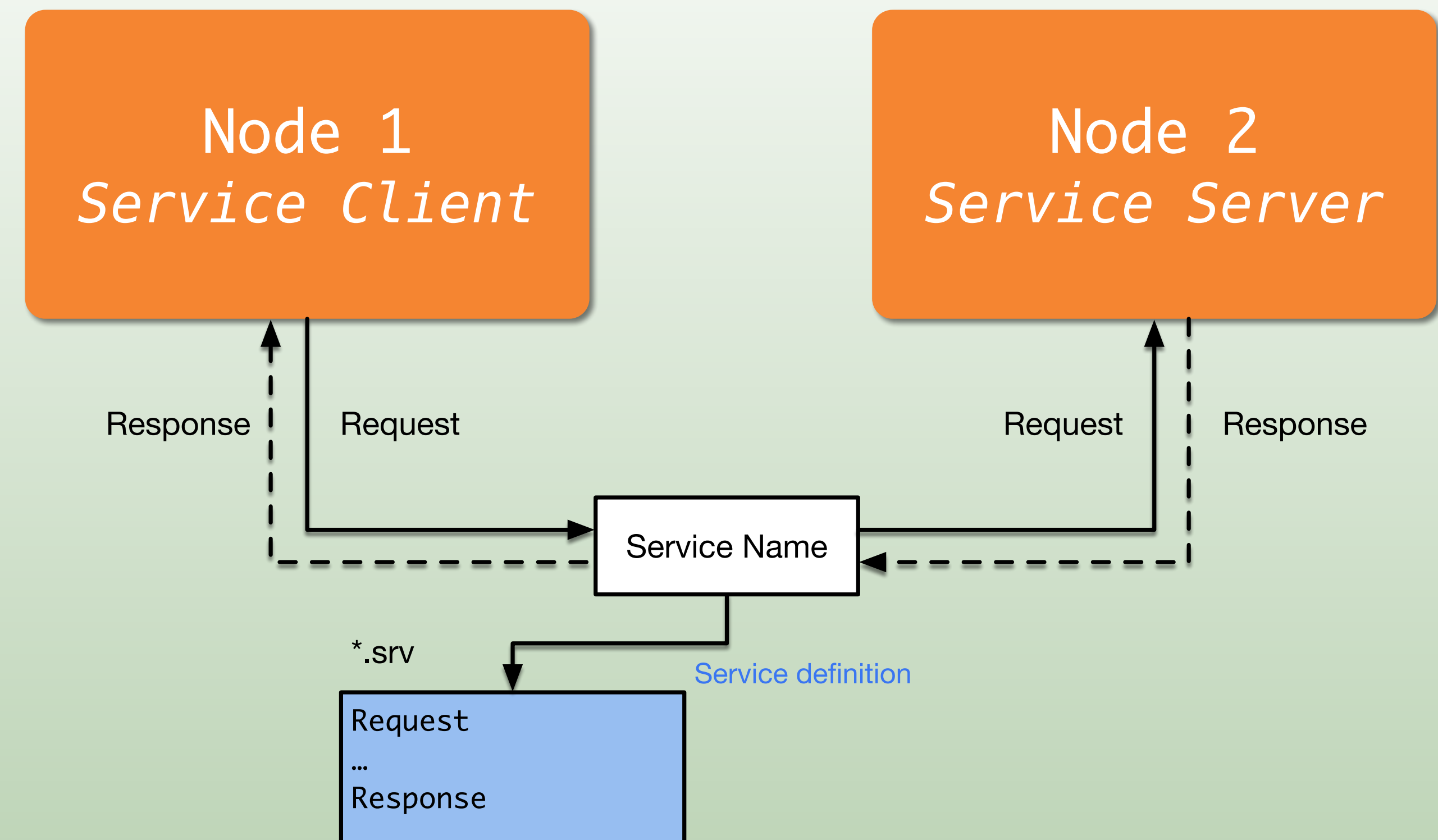
```
~$ros2 service list
```

Show service type:

```
~$ros2 service type /service_name
```

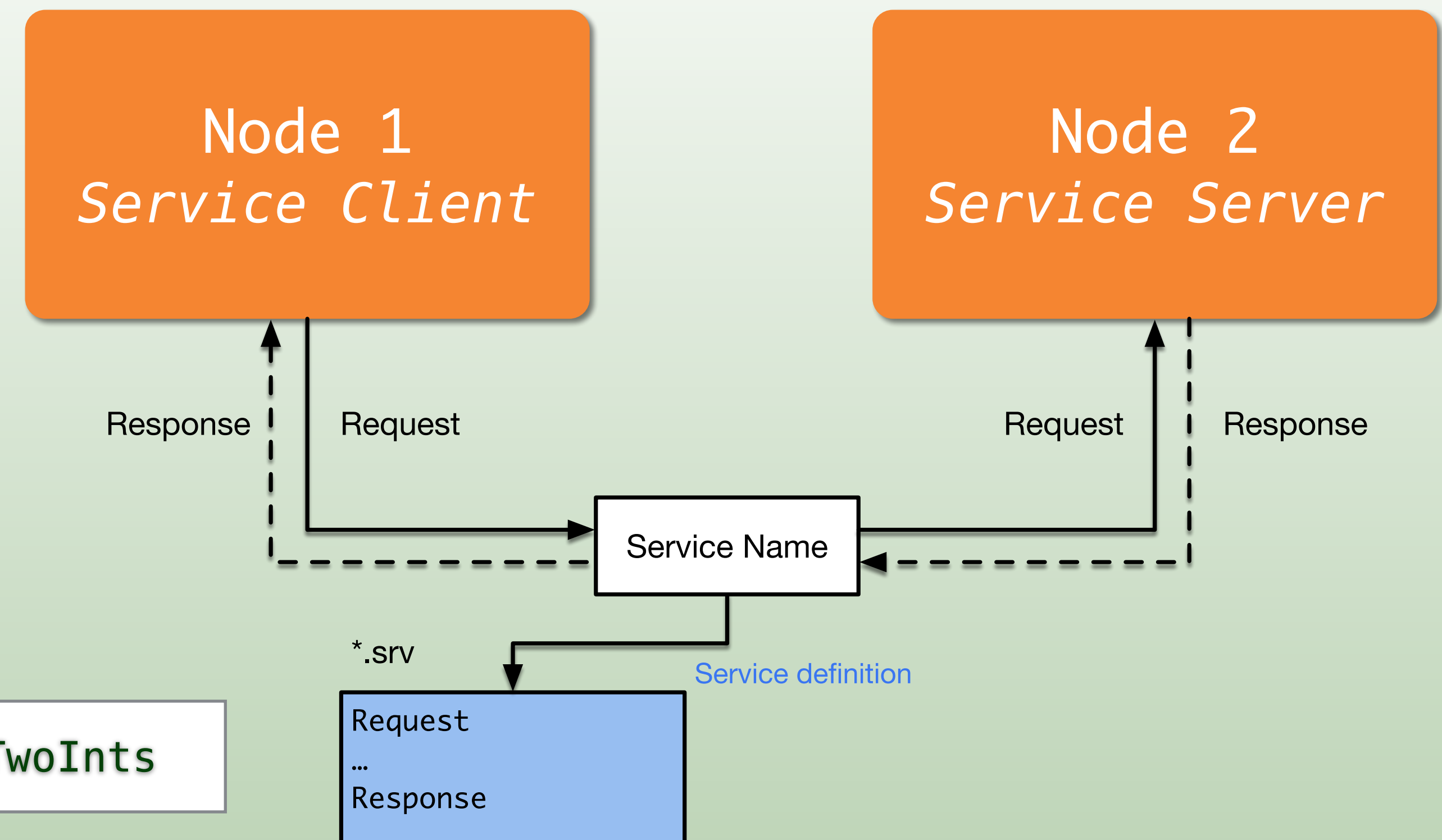
Call service requesting content:

```
~$ros2 service call <service> <service_type> <request>
```



Details at: [Services](#)

► Examples:



Show the service interface:

```
~$ros2 interface show example_interfaces/srv/AddTwoInts
```

Call the service:

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

ROS 2 SERVICE EXAMPLE

std_srvs/srv/Trigger

```
---  
bool success    # indicate successful run of triggered service  
string message  # informational, e.g. for error messages
```

Request

Response

nav_msgs/srv/GetPlan

Request

Response

```
# Get a plan from the current position to the goal Pose  
  
# The start pose for the plan  
geometry_msgs/PoseStamped start  
  
# The final pose of the goal position  
geometry_msgs/PoseStamped goal  
  
# If the goal is obstructed, how many meters the planner can  
# relax the constraint in x and y before failing.  
float32 tolerance  
---  
nav_msgs/Path plan
```

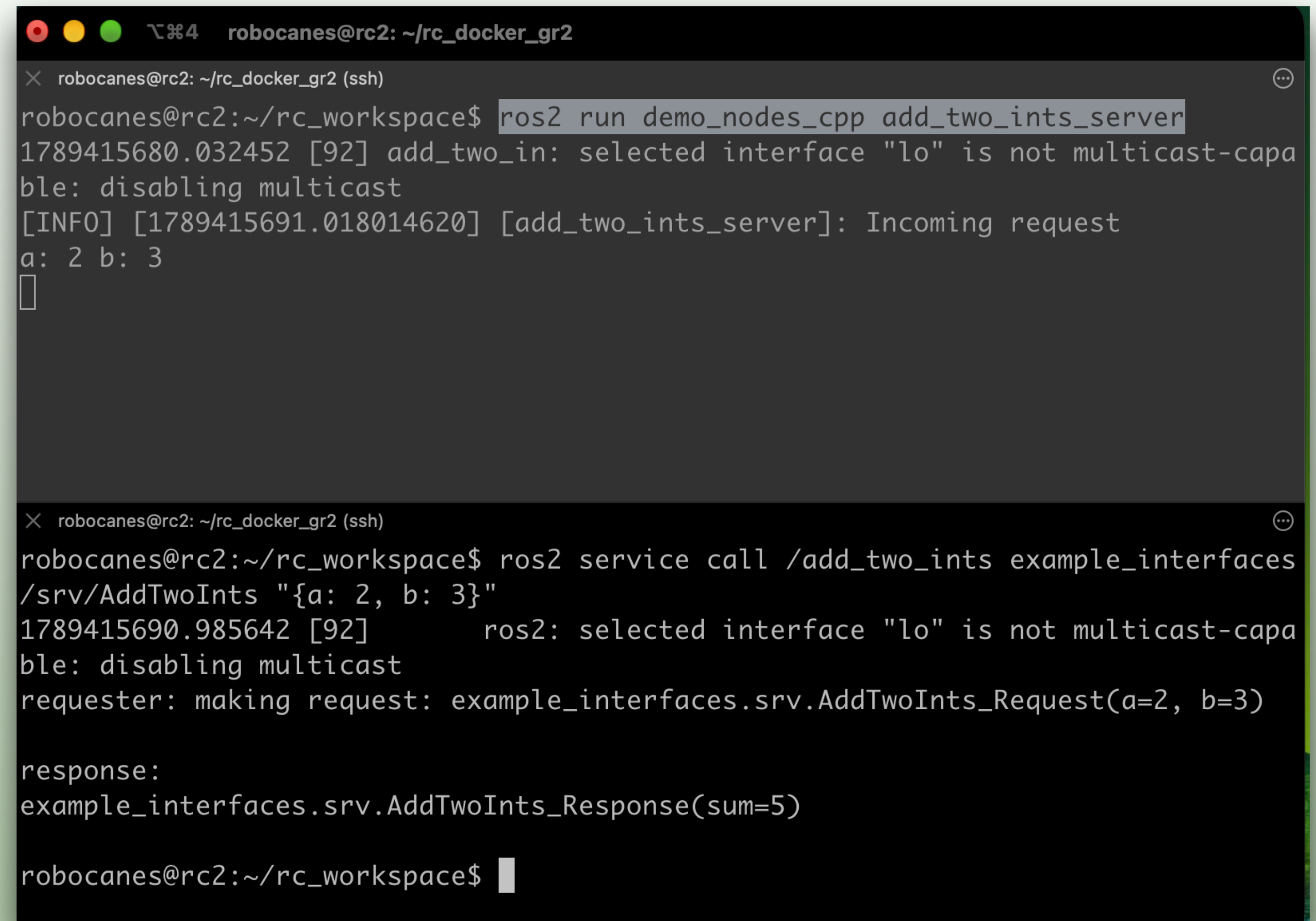
ROS 2 SERVICE EXAMPLE

T1: run service

```
~$ros2 run demo_nodes_cpp add_two_ints_server
```

T2: call service

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

A terminal window titled 'robocanes@rc2: ~/rc_docker_gr2' showing two separate terminal sessions. The top session shows the command 'ros2 run demo_nodes_cpp add_two_ints_server' being executed, with output indicating the server is running and receiving a request for a=2, b=3. The bottom session shows the command 'ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 2, b: 3}"' being executed, with output showing the request being made and the response being received: 'example_interfaces.srv.AddTwoInts_Response(sum=5)'.

```
robocanes@rc2: ~/rc_docker_gr2 (ssh)
robocanes@rc2:~/rc_workspace$ ros2 run demo_nodes_cpp add_two_ints_server
1789415680.032452 [92] add_two_in: selected interface "lo" is not multicast-capable: disabling multicast
[INFO] [1789415691.018014620] [add_two_ints_server]: Incoming request
a: 2 b: 3
█

robocanes@rc2: ~/rc_docker_gr2 (ssh)
robocanes@rc2:~/rc_workspace$ ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 2, b: 3}"
1789415690.985642 [92] ros2: selected interface "lo" is not multicast-capable: disabling multicast
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=2, b=3)

response:
example_interfaces.srv.AddTwoInts_Response(sum=5)

robocanes@rc2:~/rc_workspace$ █
```

- ▶ Create a service server with `create_service()`

```
auto service =  
    node->create_service<ServiceType>(  
        "service_name", callback);
```

- ▶ When a request is received, the callback is called
- ▶ The callback receives the **request** and a **response object**
- ▶ Fill in the response object; no true/false return value is needed
- ▶ `rclcpp::spin()` processes incoming service requests

```
#include "rclcpp/rclcpp.hpp"  
#include "example_interfaces/srv/add_two_ints.hpp"  
  
void add(  
    const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,  
    std::shared_ptr<example_interfaces::srv::AddTwoInts::Response> response)  
{  
    response->sum = request->a + request->b;  
}  
  
int main(int argc, char * argv[])  
{  
    rclcpp::init(argc, argv);  
  
    auto node = rclcpp::Node::make_shared("add_two_ints_server");  
  
    auto service =  
        node->create_service<example_interfaces::srv::AddTwoInts>(  
            "add_two_ints", add);  
  
    rclcpp::spin(node);  
  
    rclcpp::shutdown();  
    return 0;  
}
```

ROS 2 C++ CLIENT LIBRARY — SERVICE CLIENT

- Create a service client:

```
auto client =  
    node->create_client<ServiceType>("service_name");
```

- Create service request contents

```
auto request =  
    std::make_shared<ServiceType::Request>();
```

- Send request asynchronously

```
auto result =  
    client->async_send_request(request);
```

- Retrieve response from future

```
auto response = result.get();
```

Details at: [Write simple C++ service&client](#)

```
#include "rclcpp/rclcpp.hpp"  
#include "example_interfaces/srv/add_two_ints.hpp"  
  
int main(int argc, char * argv[])  
{  
    rclcpp::init(argc, argv);  
  
    auto node =  
        rclcpp::Node::make_shared("add_two_ints_client");  
  
    auto client =  
        node->create_client<example_interfaces::srv::AddTwoInts>(  
            "add_two_ints");  
  
    auto request =  
        std::make_shared<  
            example_interfaces::srv::AddTwoInts::Request>();  
  
    request->a = 2;  
    request->b = 3;  
  
    while (!client->wait_for_service(std::chrono::seconds(1))) {  
        RCLCPP_INFO(node->get_logger(),  
            "Waiting for service...");  
    }  
  
    auto future = client->async_send_request(request);  
  
    if (rclcpp::spin_until_future_complete(node, future) ==  
        rclcpp::FutureReturnCode::SUCCESS)  
    {  
        RCLCPP_INFO(node->get_logger(),  
            "Sum: %ld", future.get()->sum);  
    }  
  
    rclcpp::shutdown();  
    return 0;  
}
```

ROS 2 PYTHON CLIENT LIBRARY (RCLPY) — SERVICE SERVER

- ▶ Create a service server with `create_service()`

```
service = node.create_service(  
    AddTwoInts, 'add_two_ints', callback)
```

- ▶ When a request is received, the callback is called
- ▶ The callback receives the **request** and **response**
- ▶ Fill in the response object and return it
- ▶ `rclpy.spin(node)` processes incoming service requests

```
import rclpy  
from rclpy.node import Node  
from example_interfaces.srv import AddTwoInts  
  
def add_two_ints(request, response):  
    response.sum = request.a + request.b  
    return response  
  
rclpy.init()  
  
node = Node('add_two_ints_server')  
  
service = node.create_service(  
    AddTwoInts,  
    'add_two_ints',  
    add_two_ints)  
  
rclpy.spin(node)  
  
node.destroy_node()  
rclpy.shutdown()
```

ROS 2 PYTHON CLIENT LIBRARY (RCLPY) — SERVICE CLIENT

- ▶ Create a service client with `create_client()`

```
client = node.create_client(  
    AddTwoInts, 'add_two_ints')
```

- ▶ Wait for the service to become available

```
while not client.wait_for_service(timeout_sec=1.0):  
    node.get_logger().info('Waiting for service...')
```

- ▶ Create and fill the request object

```
request = AddTwoInts.Request()  
request.a = 2  
request.b = 3
```

- ▶ Send the request asynchronously with `call_async()`

```
future = client.call_async(request)  
rclpy.spin_until_future_complete(node, future)  
response = future.result()
```

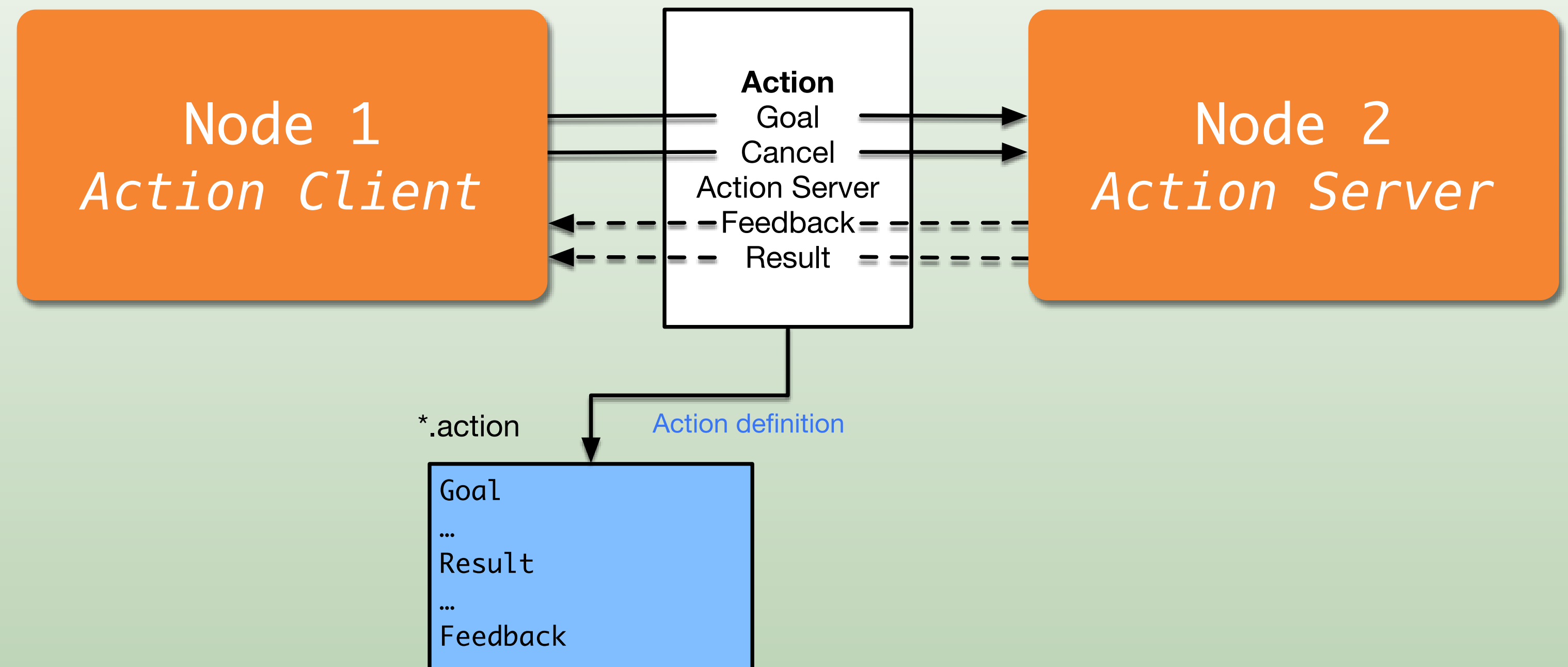
- ▶ Spin until the future completes
- ▶ Read the response from the future

```
import rclpy  
from rclpy.node import Node  
from example_interfaces.srv import AddTwoInts  
  
rclpy.init()  
  
node = Node('add_two_ints_client')  
  
client = node.create_client(  
    AddTwoInts, 'add_two_ints')  
  
while not client.wait_for_service(timeout_sec=1.0):  
    node.get_logger().info('Waiting for service...')  
  
request = AddTwoInts.Request()  
request.a = 2  
request.b = 3  
  
future = client.call_async(request)  
  
rclpy.spin_until_future_complete(node, future)  
  
response = future.result()  
  
node.get_logger().info(  
    f'Sum: {response.sum}')
```

```
node.destroy_node()  
rclpy.shutdown()
```

ROS 2 ACTIONS

- ▶ Similar to services, but intended for **long-running, goal-oriented tasks**
- ▶ Actions allow the client to:
 - ▶ send a **goal**
 - ▶ receive **feedback** during execution
 - ▶ **cancel** a goal
 - ▶ receive the final **result**
- ▶ Well suited for tasks such as navigation, manipulation, or trajectory execution
- ▶ Actions are defined in `.action` **interface files**
- ▶ An action definition contains:
 - ▶ Goal
 - ▶ Result
 - ▶ Feedback



Averaging.action

```
#goal definition
int32 samples
---
#result definition
float32 mean
float32 std_dev
---
#feedback
int32 sample
float32 data
float32 mean
float32 std_dev
```

← Goal

← Result

← Feedback

nav2_msgs/action/FollowPath

Goal → nav_msgs/Path path string
controller_id string goal_checker_id

Result → std_msgs/Empty result

Feedback → float32 distance_to_goal
float32 speed

COMPARISON OF ROS 2 COMMUNICATION OPTIONS

	Parameters	Topics	Services	Actions
Description	Node-specific configuration values, readable/changeable at runtime	Continuous asynchronous data streams	Request/response interaction	Long-running goal-oriented tasks with feedback and cancellation
Application	Configuration and tuning	Continuous data flow	Short operations, queries, triggers	Task execution
Examples	Camera settings, thresholds, controller gains	Sensor data, robot state, commands	Reset system, compute value, request state	Navigation, manipulation, trajectory execution

- ▶ ROS 2 normally uses the system clock as its time source
- ▶ For simulation or playback of recorded data, ROS 2 can use simulated time
- ▶ Simulated time is provided on the /clock topic
- ▶ Each node that should use simulated time must have:
 - ▶ ROS 2 uses a node-local parameter named `use_sim_time`:

```
~$ros2 param set /node_name use_sim_time true
```
 - ▶ Simulation systems such as Gazebo or Isaac Sim can publish /clock
 - ▶ `ros2 bag play` can also publish simulated time when configured for clock playback

- ▶ To take advantage of the simulated time, you should always use the ROS Time APIs:

`rclcpp::Time`

```
rclcpp::Time begin = node->now();
```

`rclcpp::Duration`

```
rclcpp::Duration duration =  
    rclcpp::Duration::from_seconds(0.5);
```

`rclcpp::Rate`

```
rclcpp::Rate rate(10.0);
```

- ▶ `rclcpp::Clock` supports different clock types
- ▶ ROS time, system time, and steady time are available
- ▶ Use **`node->now()`** when you want the node's configured ROS time source

ROS 2 BAGS (ROSBAG2)

- ▶ rosbag2 records and stores ROS 2 topic data
- ▶ Useful for logging, replay, visualization, and offline analysis
- ▶ A recording is usually stored in a directory
- ▶ In ROS 2 Humble, the default storage backend is typically SQLite3
- ▶ Recording also contains metadata such as topic types and duration

Record topics in a bag (Ctrl-C to stop)

```
~$ros2 bag record -a
```

Record specific topics

```
~$rosvbag record /topic_1 /topic_2 /topic_3
```

Show information about a bag

```
~$ros2 bag info bag_name
```

Read a bag and play/publish its contents

```
~$ros2 bag play bag_name
```

Including playback options, e.g.

```
~$ros2 bag play bag_name --rate 0.5
```

--loop

Look playback

ROS 2 uses rosbag2; recordings are typically directories containing the storage file(s) and metadata.yaml

Details at:
[Recording and playing back data](#)

DEBUGGING STRATEGIES

- ▶ Build and run code often to catch problems early
- ▶ Read compiler and runtime error messages carefully
- ▶ Check ROS 2 data flow with:
 - ▶ `ros2 node list`
 - ▶ `ros2 node info`
 - ▶ `ros2 topic list`
 - ▶ `ros2 topic echo`
 - ▶ `ros2 topic hz`
 - ▶ `rqt_graph`
- ▶ Visualize and inspect data with:
 - ▶ RViz2
 - ▶ `rqt_plot`
 - ▶ `rqt_console`
- ▶ Add logging at intermediate steps:
 - ▶ `RCLCPP_INFO`, `RCLCPP_DEBUG`
 - ▶ `node.get_logger().info()`, `.debug()`
Check arguments, return values, and exceptions
 - ▶ Use breakpoints if you use an IDE
 - ▶ Maintain code with unit tests and integration tests

Build with DEBUG mode and use GDB and/or Valgrind

```
~$colcon build --cmake-args -DCMAKE_BUILD_TYPE=Debug
```

```
~$rm -rf build install log && colcon build
```

FURTHER REFERENCES

- ▶ ROS 2 Documentation
 - ▶ <https://docs.ros.org/en/humble/>
- ▶ Installation
 - ▶ <https://docs.ros.org/en/humble/Installation.html>
- ▶ Tutorials
 - ▶ <https://docs.ros.org/en/humble/Tutorials.html>
- ▶ Packages
 - ▶ <https://index.ros.org/>
- ▶ Services / Actions
 - ▶ <https://docs.ros.org/en/humble/Concepts/Basic/About-Services.html>
 - ▶ <https://docs.ros.org/en/humble/Concepts/Basic/About-Actions.html>
- ▶ rosbag2
 - ▶ <https://docs.ros.org/en/humble/Tutorials/Ros2bag/Recording-And-Playing-Back-Data.html>
- ▶ Debugging / CLI tools
 - ▶ <https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools.html>

Material is based on ROS 2 Humble Documentation