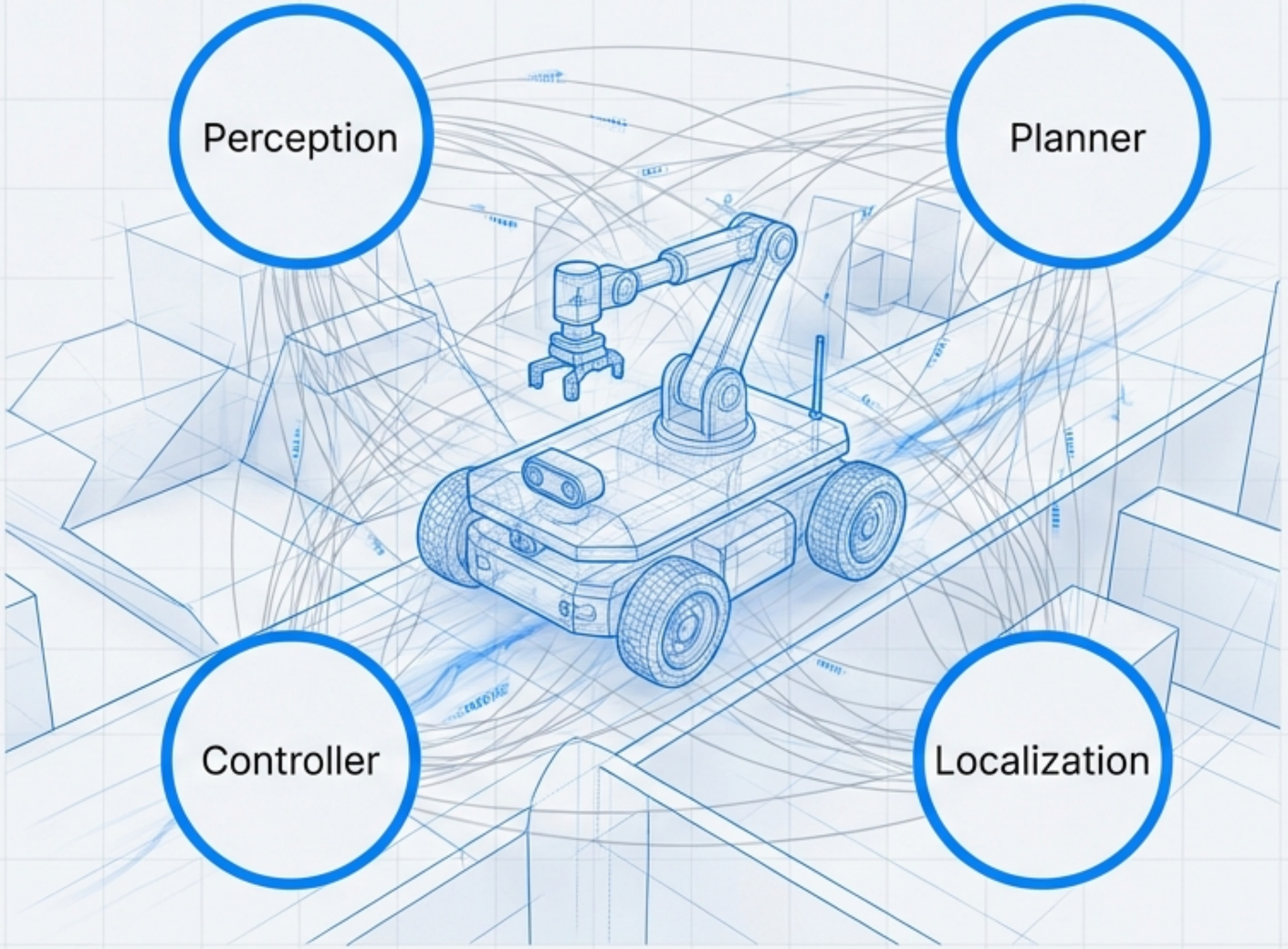
The Robotics Architect’s Toolbox

Mastering Communication in ROS2

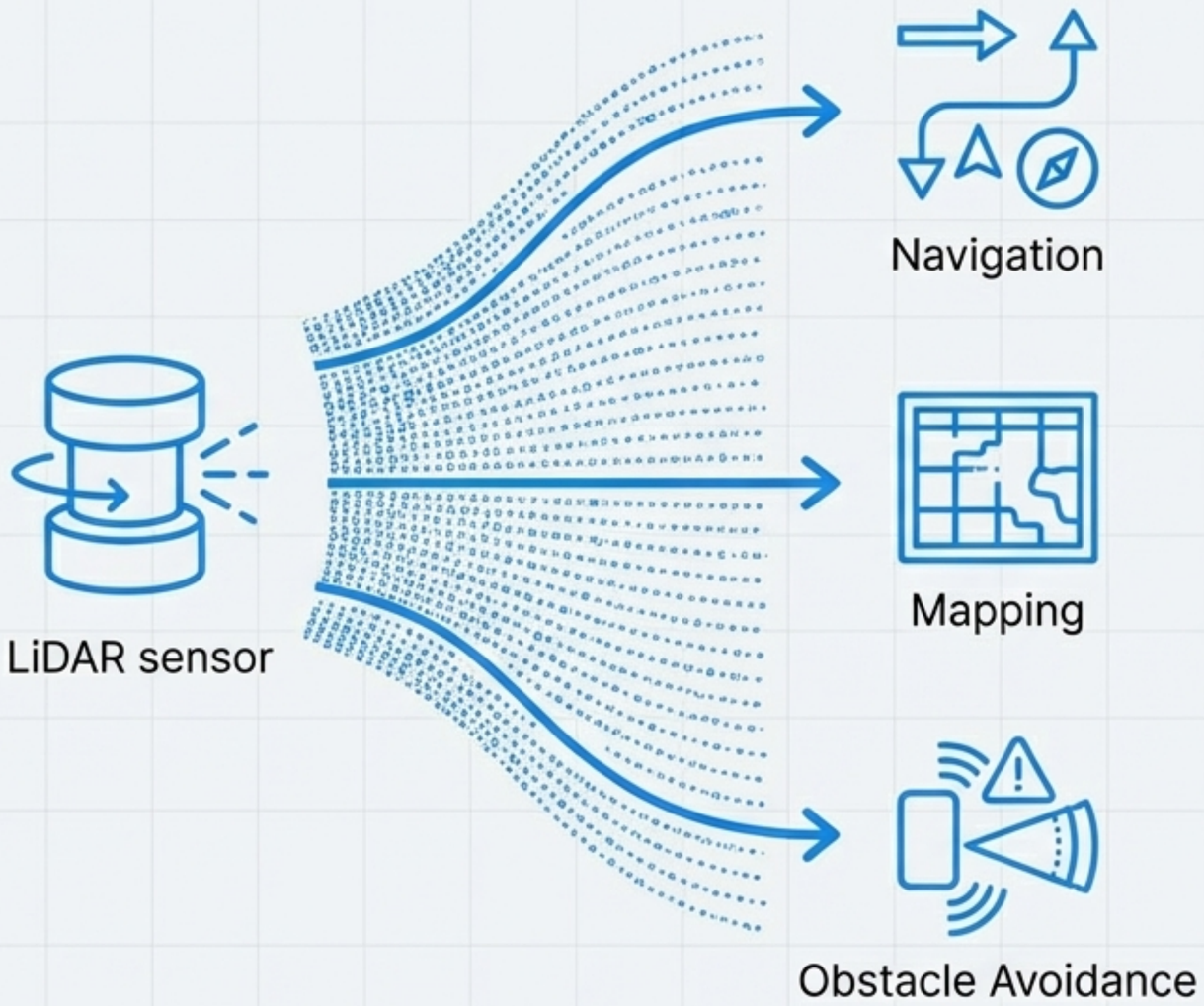
How do you conduct an orchestra of code?

An autonomous robot isn’t one program. It’s dozens of independent nodes for perception, planning, control, and localization, all running at once. How do we make them communicate seamlessly to achieve a single, complex goal?



Challenge #1: The Constant Stream

Your robot's LiDAR spins, generating thousands of points per second. The navigation, obstacle avoidance, and mapping nodes **all** need this data, simultaneously and continuously. How do you broadcast this information efficiently without coupling the systems together?

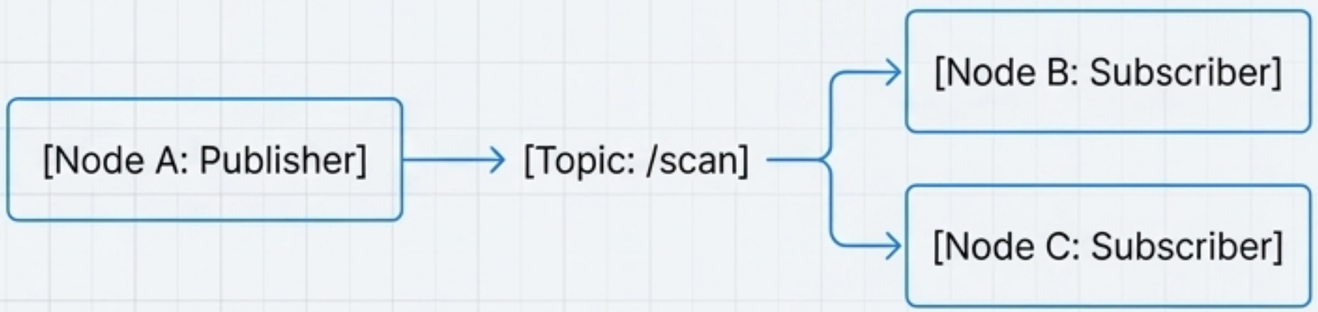


The Solution: Publishers & Subscribers

This is a one-to-many broadcast system.

A Publisher sends data on a named channel (a Topic) without knowing who, if anyone, is listening. Subscribers tune into that topic to receive the data.

It's asynchronous and decoupled—the perfect “fire-and-forget” model for continuous data streams.



Quality of Service (QoS) Note
You can configure topics to be “Reliable” (guaranteed delivery, like TCP) or “Best Effort” (faster, but packets can be dropped, like UDP).

The Blueprint: Creating a Publisher

```
// C++ Publisher for Robot Velocity
class SimplePublisher : public rclcpp::Node
{
public:
  SimplePublisher() : Node("simple_publisher")
  {
    // 1. Create the publisher on the '/cmd_vel' topic
    publisher_ = this->create_publisher<geometry_msgs::msg::Twist>("/cmd_vel", 10);

    // 2. Set up a timer to call the publish function
    timer_ = this->create_wall_timer(500ms, std::bind(&SimplePublisher::timer_callback, this));
  }
private:
  void timer_callback()
  {
    // 3. Create the message
    auto message = geometry_msgs::msg::Twist();
    message.linear.x = 0.5;
    message.angular.z = 0.5;

    // 4. Publish the message
    publisher_->publish(message);
  }
  rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr publisher_;
  rclcpp::TimerBase::SharedPtr timer_;
};
```

1. **create_publisher**: Declares the topic name and message type.
2. **create_wall_timer**: Sets the publishing frequency.
3. **geometry_msgs::msg::Twist**: Defines the data structure.
4. **publish()**: The command that sends the data.

Challenge #2: The Direct Command

Your robot needs to perform a discrete, blocking action. For example, triggering a camera to save a single image to disk. You don't just want to send the command; you need to *wait* and get a direct confirmation back: “Success, the image is saved at `/tmp/image.jpg`.”
How do you handle this tight request/response loop?

The Solution: Services

This is a one-to-one, synchronous request/response model. A **Client** sends a request and *waits* (blocks) until the **Server** processes it and returns a single response. It's a remote procedure call, ideal for transactions or commands that must be confirmed before the program proceeds.

The Blueprint: Calling a Service

```
// C++ Service Client Example
// 1. Create a client for the 'trigger_camera' service
client = this->create_client<example_interfaces::srv::Trigger>("trigger_camera");

// 2. Wait for the service to be available
while (!client->wait_for_service(1s)) {
   RCLCPP_INFO(this->get_logger(), "service not available, waiting again...");
}

// 3. Create the request object
auto request = std::make_shared<example_interfaces::srv::Trigger::Request>();
// (Set request fields if any)

// 4. Send the request and wait for the response
auto result_future = client->async_send_request(request);
if (rclcpp::spin_until_future_complete(this->get_node_base_interface(), result_future) ==
   rclcpp::FutureReturnCode::SUCCESS)
{
   RCLCPP_INFO(this->get_logger(), "Success: %s", result_future.get()->message.c_str());
}
```

1. `create_client`: Specifies the service name and type.

2. `wait_for_service`: Ensures the server is running.

3. `Request`: The data being sent.

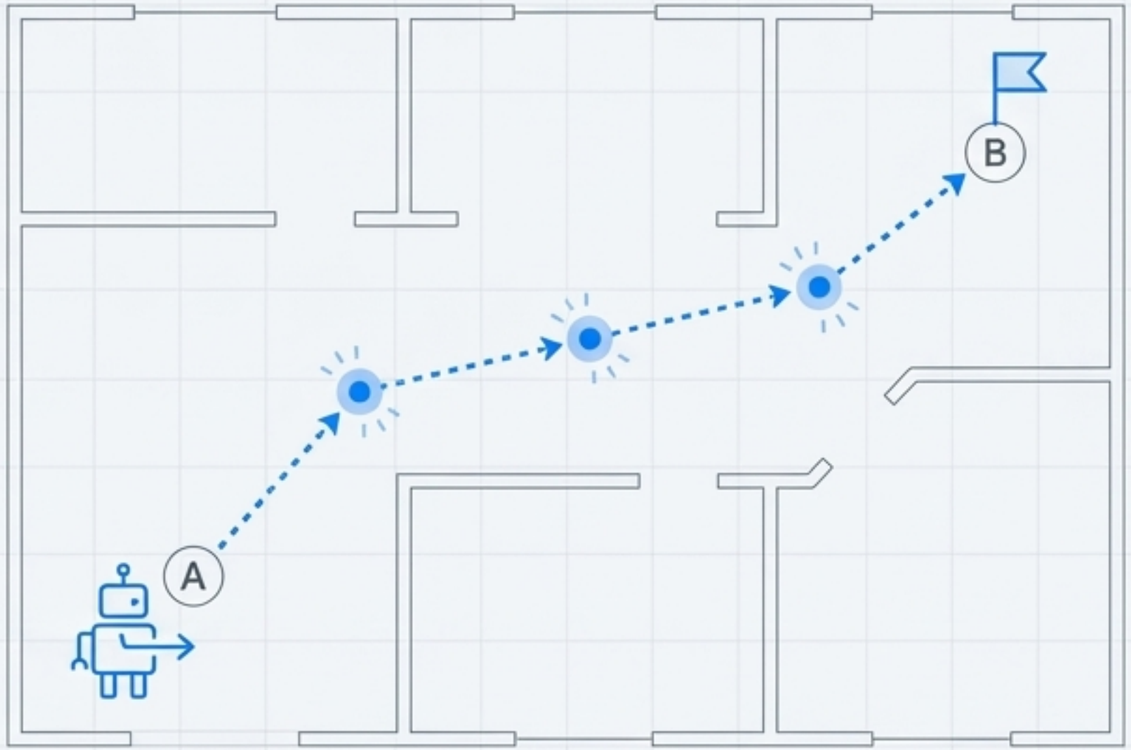
4. The core logic that sends the request and blocks until the response arrives.

Challenge #3: The Long-Running Task

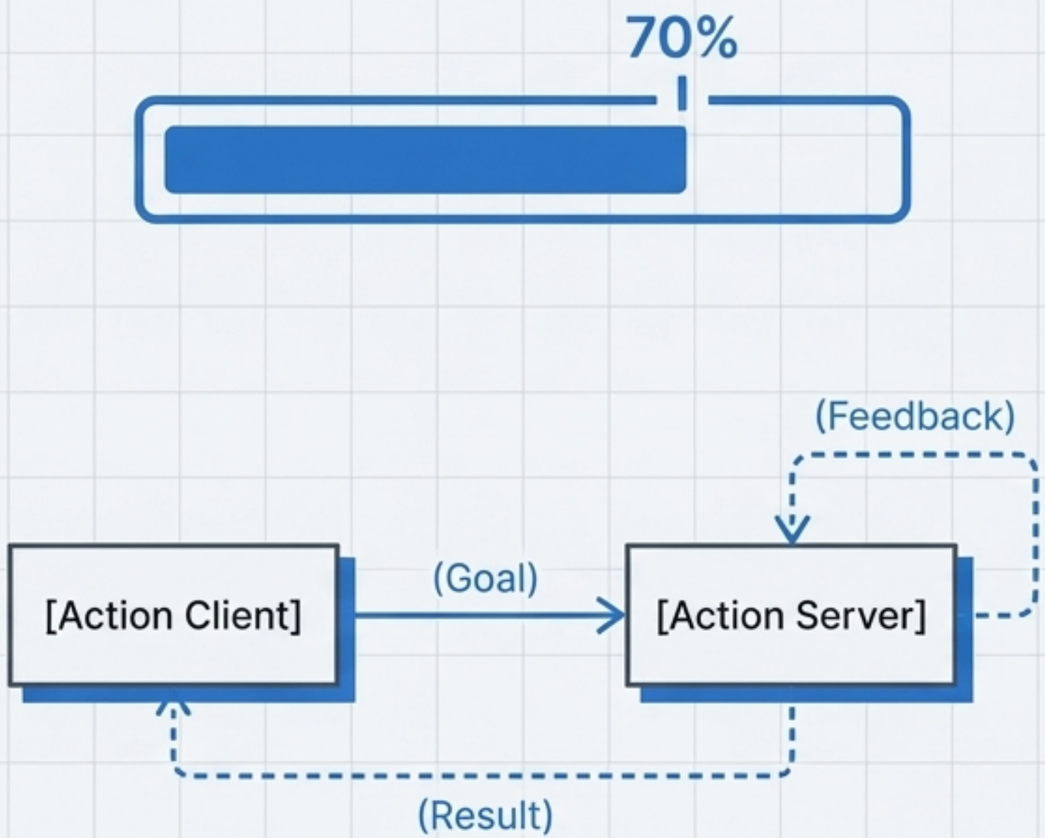
You want your mobile robot to navigate to a goal 50 meters away. This could take 30 seconds or more.
A Service call would block your entire control node for the whole duration, which is unacceptable.

You need to initiate the task, monitor its progress along the way, and receive a final result when it’s done.

How do you manage this?



The Solution: Actions



Actions are for long-running, asynchronous goals that provide continuous feedback and can be preempted.
Unlike Services, an Action client doesn’t block. It sends a goal, gets feedback periodically, and receives a final result via callbacks.

It’s the ideal pattern for navigation, manipulation, or any task with distinct phases.

The Action Lifecycle: Goal, Feedback, Result

GOAL

The client sends a request to start the task (e.g., “Navigate to coordinates X, Y”). The server accepts or rejects it.

FEEDBACK

While the task is executing, the server sends a stream of updates (e.g., “Current distance to goal: 25m... 20m... 15m...”).

RESULT

Once the task is complete, the server sends one final message with the outcome (e.g., “Success: Goal reached”).

The Blueprint: Executing an Action

```cpp
// C++ Action Client Example
using NavigateToPose = nav2_msgs::action::NavigateToPose;

// 1. Create the Action Client
this->action_client_ = rclcpp_action::create_client<NavigateToPose>(this, "navigate_to_pose");

// 2. Define callbacks for feedback and result
auto send_goal_options = rclcpp_action::Client<NavigateToPose>::SendGoalOptions();
send_goal_options.feedback_callback =
    std::bind(&MyNode::feedback_callback, this, _1, _2);
send_goal_options.result_callback =
    std::bind(&MyNode::result_callback, this, _1);

// 3. Create the goal message
auto goal_msg = NavigateToPose::Goal();
goal_msg.pose.header.frame_id = "map";
goal_msg.pose.pose.position.x = 1.0;

// 4. Send the goal asynchronously
this->action_client_->async_send_goal(goal_msg, send_goal_options);
```

1. `create_client`: Establishes the connection to the Action server.

2. Shows how separate functions are registered to handle in-progress updates and the final outcome.

3. `Goal()`: The data defining the task.

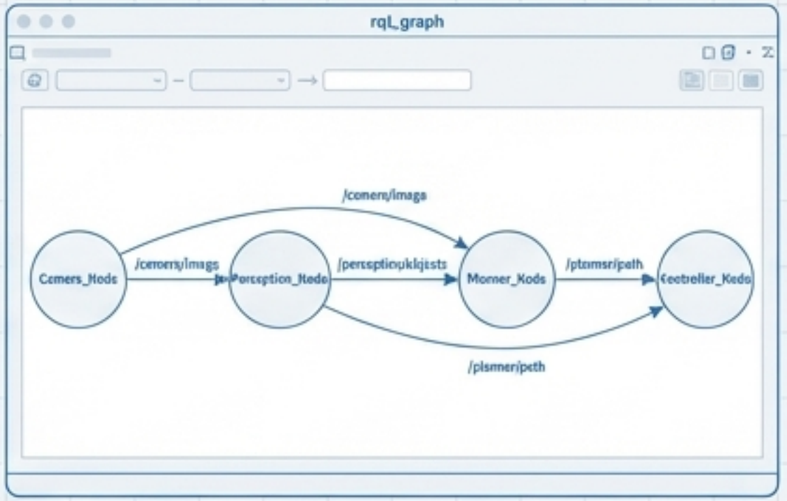
4. `async_send_goal`: The non-blocking call that starts the action.

Choosing the Right Tool for the Job

Feature	 Pub/Sub (Topics)	 Services	 Actions
Use Case	Continuous data streams (sensors), status updates	Triggering discrete tasks, requesting data	Commanding long-running, complex behaviors
Communication	One-to-Many, Decoupled	One-to-One	One-to-One
Execution	Asynchronous (“Fire & Forget”)	Synchronous (Client blocks until response)	Asynchronous (Non-blocking with feedback)
Analogy	Megaphone Broadcast	Walkie-Talkie Call	Project Plan with Milestones

Your First Steps as an Architect

1. **Start with the Problem**: Before writing code, ask: Is it a data stream, a quick command, or a long task? Let the problem define the tool.
2. **Introspect Your System**: Use the command line to see what's happening.
   - **`ros2 topic list`**: See all active data streams.
   - **`ros2 topic echo /topic_name`**: Listen in on a specific topic's data.
   - **`ros2 topic info /topic_name`**: See who is publishing and subscribing.
3. **Visualize the Architecture**: Use `rqt_graph` to generate a live diagram of your nodes and how they're connected. It's the single best tool for debugging communication issues.




```
$ ros2 topic list
/camera/image_raw
/camera/camera_info
/perception/objects_detected
/planner/path_plan
/controller/velocity_cmd
/tf
/tf_static
/rosout
```

From Chaos to Cohesion

Pub/Sub, Services, and Actions are the fundamental patterns for building robust, modular, and sophisticated robot behaviors. By mastering these tools, you can transform a complex web of independent nodes into a single, cohesive system. You are the architect.