



## ROS-Industrial Basic Developer's Training Class

October 2024

Southwest Research Institute





Session 2:

# ROS Basics Continued

Southwest Research Institute





# Outline



- Services
- *Actions*
- Launch Files
- Parameters

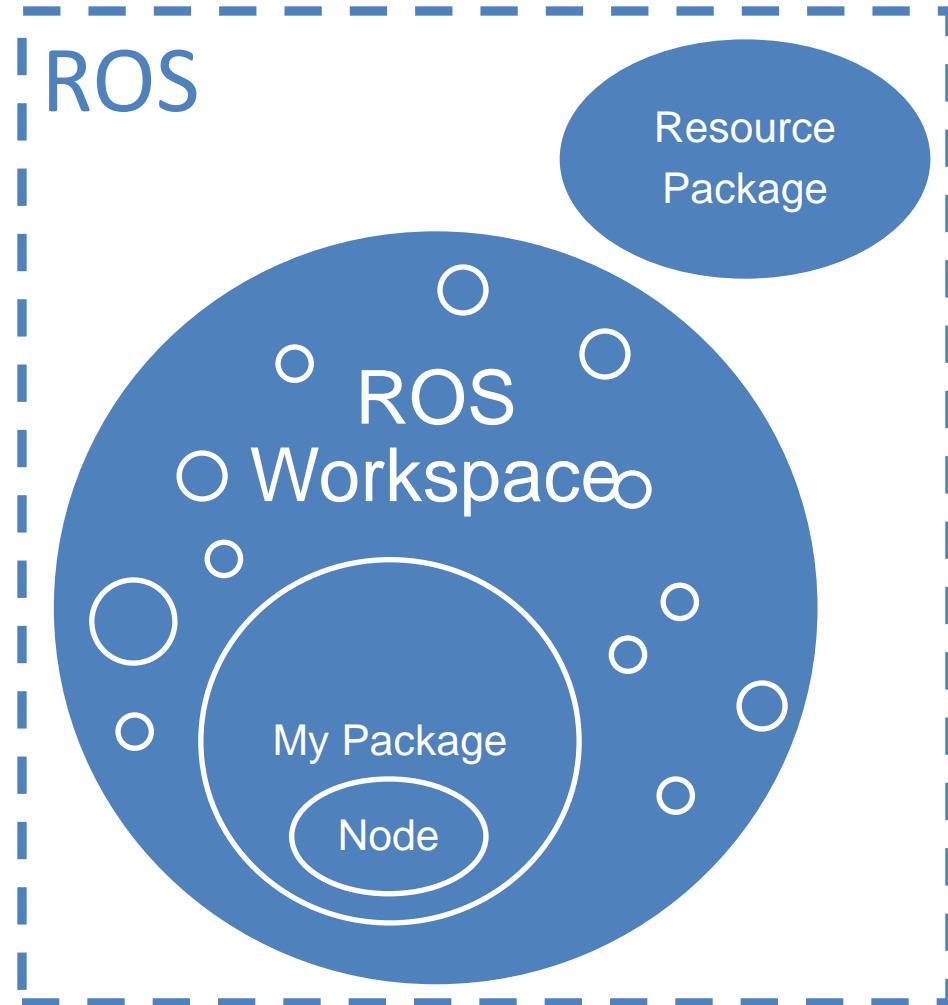




# Day 1 Progression



- ✓ Install ROS
- ✓ Create Workspace
- ✓ Add “resources”
- ✓ Create Package
- ✓ Create Node
  - ✓ Basic ROS Node
  - ✓ Interact with other nodes
    - ✓ Messages
    - ☐ Services
- ✓ Run Node
  - ✓ `ros2 run`
  - ☐ `ros2 launch`





# Services





# Services : Overview

Services are like **Function Calls**

Client



Request

Joint Pos: [J1, J2, ...]

Response

ToolPos: [X, Y, Z, ...]

Server





# Services: Details



- Each Service is made up of 2 components:
  - *Request*: sent by **client**, received by **server**
  - *Response*: generated by **server**, sent to **client**
- In ROS1, the client **blocks** when calling a service
  - In ROS2 Service Calls can be **Asynchronous**, so don't have to wait
  - Separate connection for each service call
- Typical Uses:
  - Algorithms: kinematics, perception
  - Closed-Loop Commands: move-to-position, open gripper





# Services: Syntax



- Service **definition**

- Defines Request and Response **data types**
  - *Either/both data type(s) may be **empty**. Always receive “completed” handshake.*
- Auto-generates C++ Class files (.hpp/.cpp), Python, etc.

LocalizePart.srv

Comment	→	#Localize Part
Request Data	→	string base_frame
Divider	→	---
Response Data	→	geometry_msgs/Pose
		pose



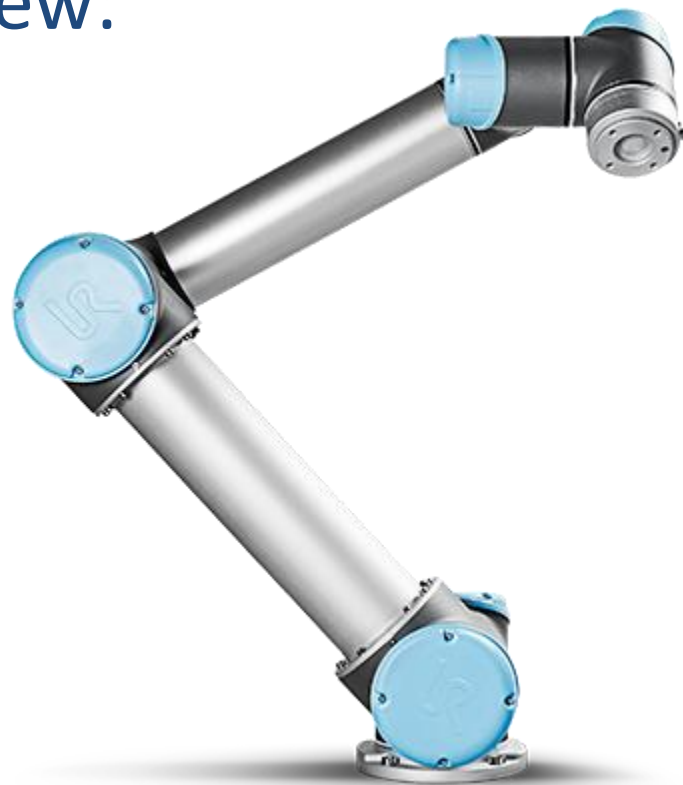




# “Real World” – Services



- Use *rqt\_srv* / *rqt\_msg* to view:
  - moveit\_msgs/GetPositionIK
  - rcl\_interfaces/GetParameters
  - moveit\_msgs/GetMotionPlan





# Services: Syntax



- **Service Server**

- Defines associated **Callback Function**
- Advertises available service (*Name, Data Type*)

Callback Function

Request Data (IN)

Response Data (OUT)

```
void findPart(LocalizePart::Request::SharedPtr req,  
             LocalizePart::Response::SharedPtr res) {  
    res->pose = lookup_pose(req->base_frame);  
}  
  
auto service = node->create_service<LocalizePart>("find_box", findPart);
```

Server Object

Service Type

Service Name

Callback Ref





# Services: Syntax



- **Service Client**
  - Connects to specific Service (*Name / Data Type*)
  - Fills in Request data
  - Calls Service

Client Object

Service Type

Service Name

```
auto client = node->create_client<LocalizePart>("find_box");
```

```
auto req = make_shared<LocalizePart::Request>();  
req->base_frame = "world";
```

← Service Request

```
auto future = client->async_send_request(req);  
rclcpp::spin_until_future_complete(node, future);
```

← Call Service  
(and wait for response)

```
auto resp = future.get();  
RCLCPP_INFO_STREAM(node->get_logger(), "Response: " << resp);
```

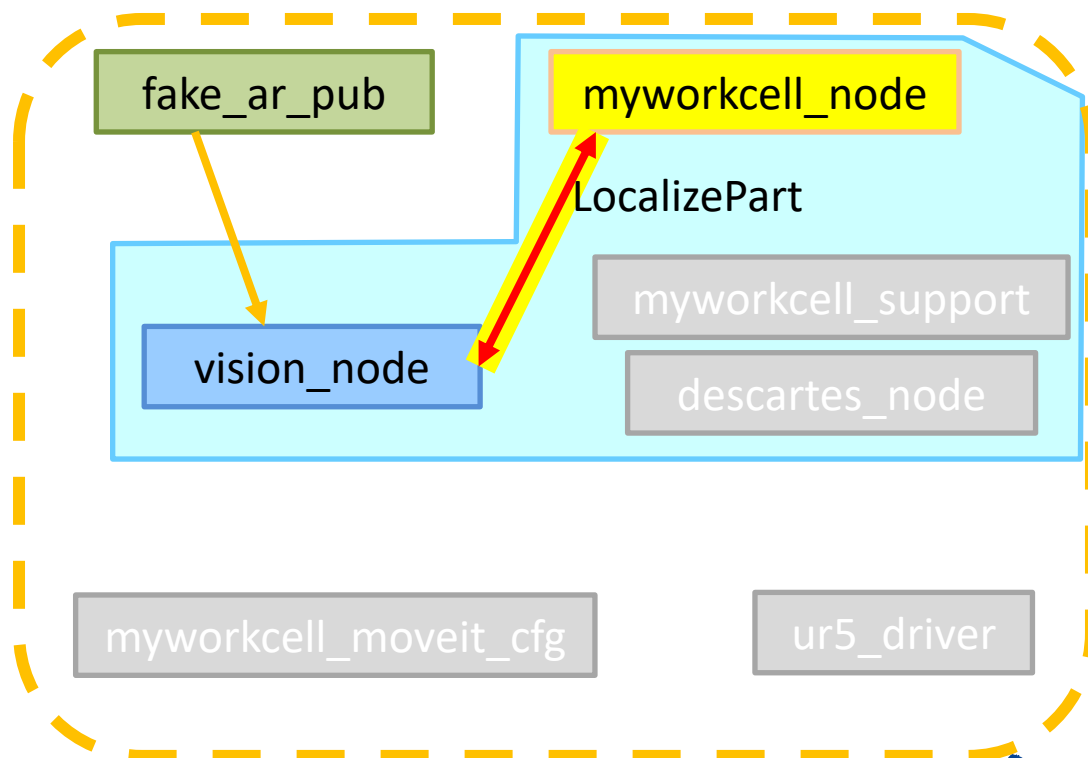
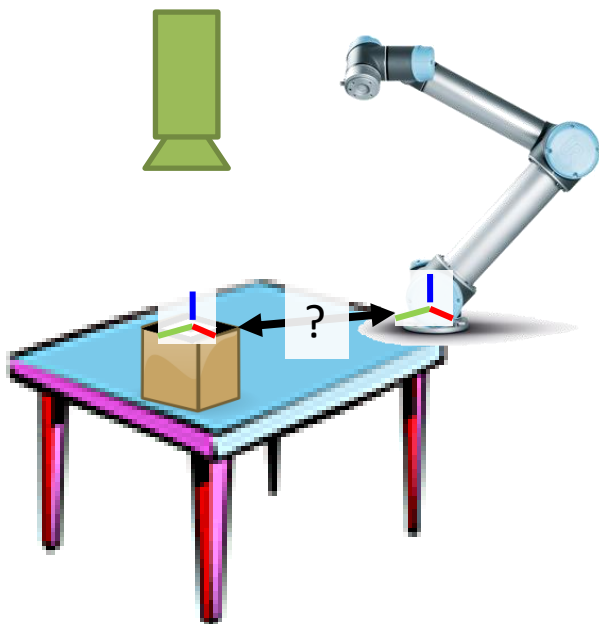
← Service Response





## Exercise 2.0

### *Creating and Using a Service*

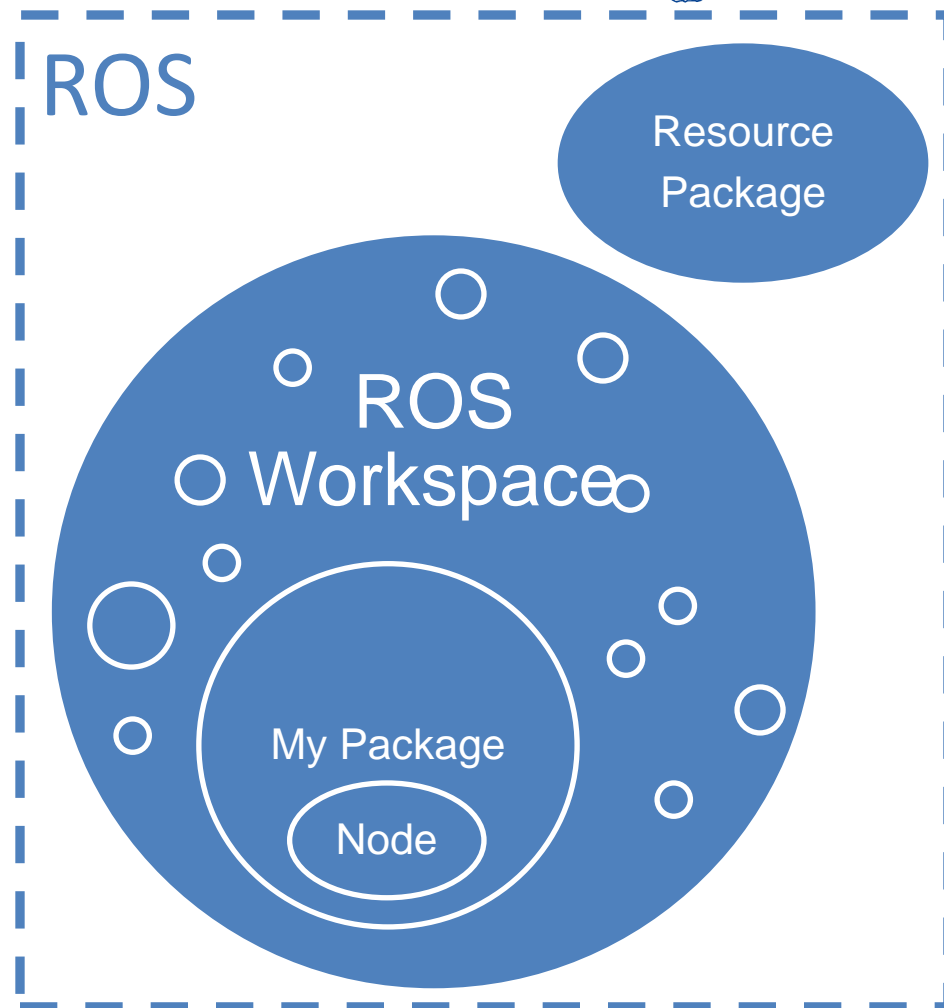




# Day 1 Progression



- ✓ Install ROS
- ✓ Create Workspace
- ✓ Add “resources”
- ✓ Create Package
- ✓ Create Node
  - ✓ Basic ROS Node
  - ✓ Interact with other nodes
    - ✓ Messages
    - ✓ Services
- ✓ Run Node
  - ✓ `ros2 run`
  - ☐ `ros2 launch`





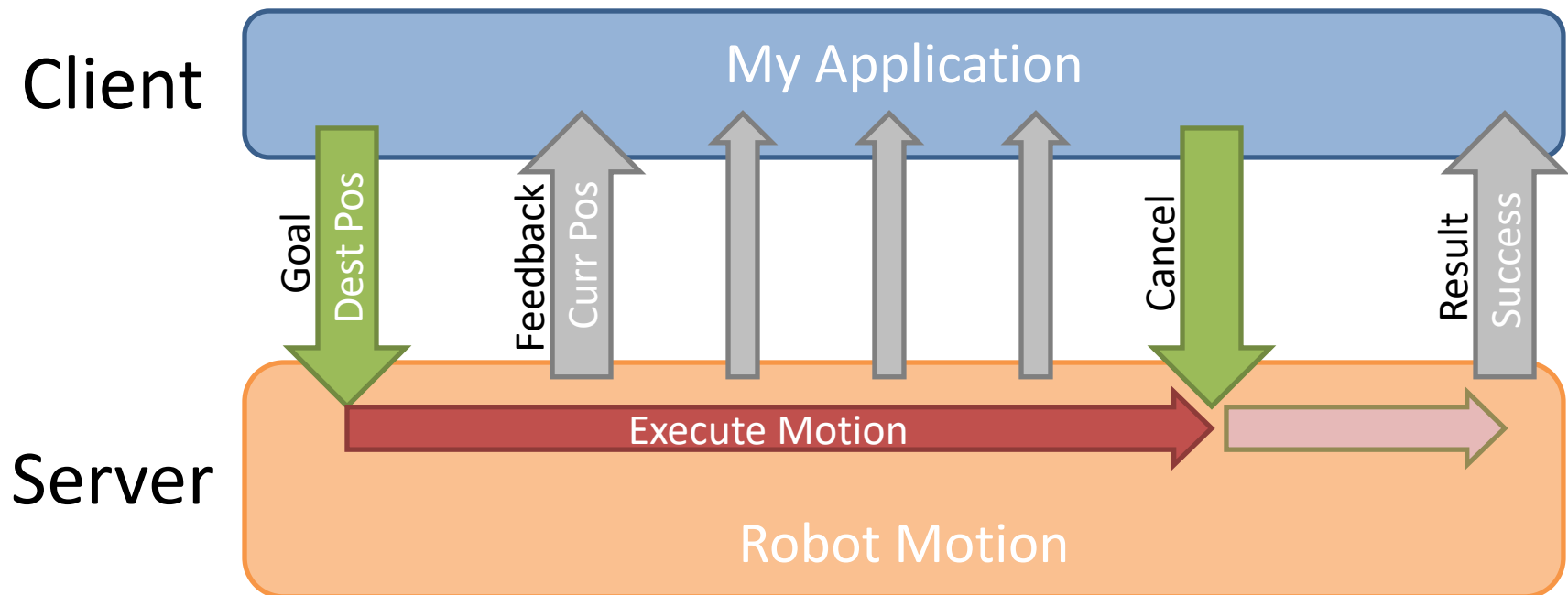
# Actions





# Actions : Overview

Actions manage **Long-Running Tasks**





# Actions: Detail



- Each action is made up of 3 components:
  - *Goal*, sent by **client**, received by **server**
  - *Result*, generated by **server**, sent to **client**
  - *Feedback*, generated by **server**
- Non-blocking in client
  - Can **monitor feedback** or **cancel** before completion
- Typical Uses:
  - “Long” Tasks: Robot Motion, Path Planning
  - Complex Sequences: Pick Up Box, Sort Widgets







# Actions: Syntax



- **Action definition**

- Defines Goal, Feedback and Result **data types**
  - Any data type(s) may be **empty**. Always receive handshakes.
- Auto-generates C++ Class files (.h/.cpp), Python, etc.

`FollowJointTrajectory.action`

Goal Data →

```
# Command Robot Motion
traj_msgs\JointTrajectory trajectory
```

```
---
```

Result Data →

```
int32 error_code
string error_string
```

```
---
```

Feedback Data →

```
uint8 status
traj_msgs\JointTrajectoryPoint actual
```





# “Real World” – Actions



- FollowJointTrajectoryAction
  - command/monitor robot trajectories
  - use rqt\_msg to view Goal, Result, Feedback
- Should be an Action...
  - GetMotionPlan
- Should not be an Action...
  - GripperCommandAction

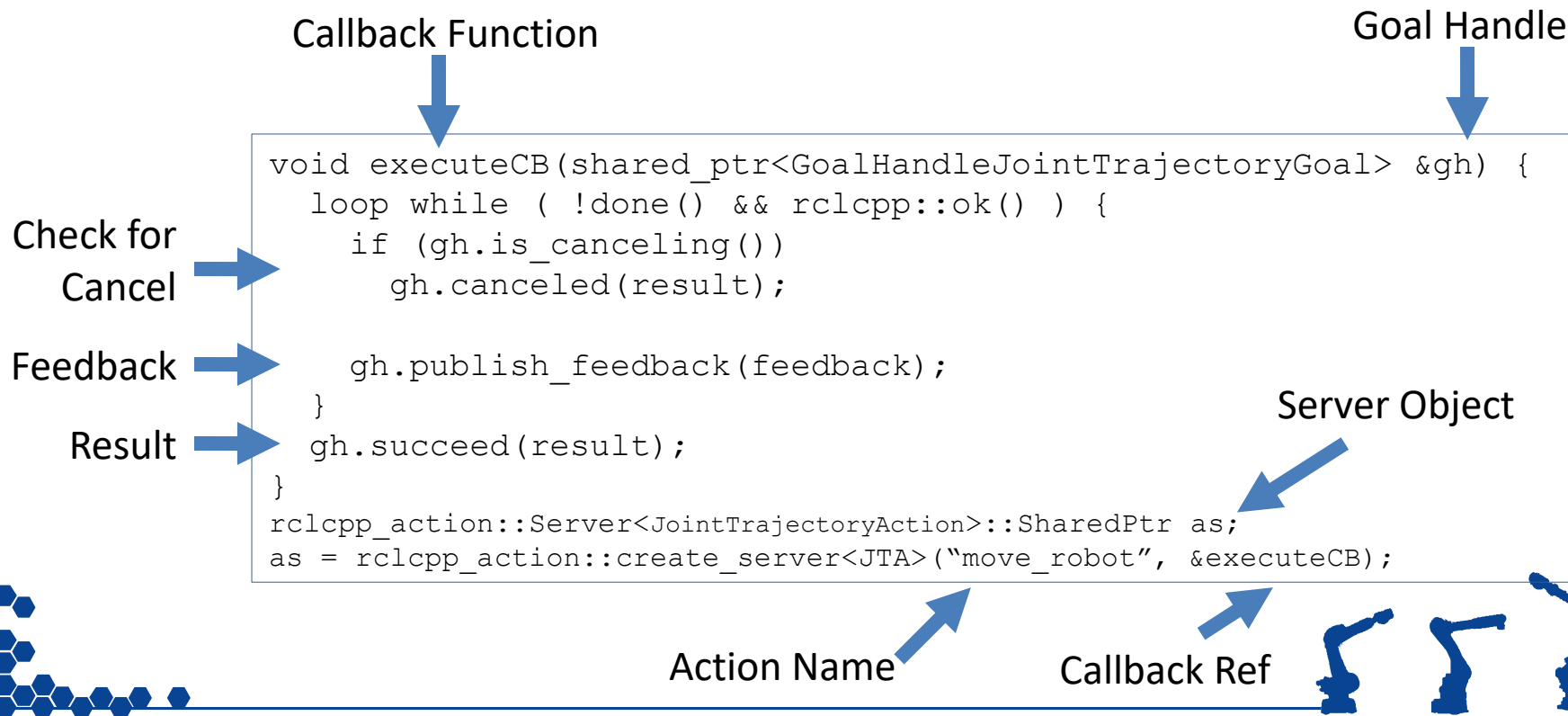




# Action Server: Syntax



- **Action Server**
  - Defines **Execute Callback**
  - Periodically **Publish Feedback**
  - Advertises available action (*Name, Data Type*)





# Action Client: Syntax



- **Action Client**

- Connects to specific Action (*Name / Data Type*)
- Fills in Goal data
- Initiate Action / Waits for Result

Action Type

Client Object

```
rclcpp_action::Client<JointTrajectoryAction>::SharedPtr ac;  
ac = rclcpp_action::create_client<JTA>("move_robot");
```

```
auto goal = JointTrajectoryAction::Goal();  
goal.trajectory = <sequence of points>;
```

```
auto future = ac->async_send_goal(goal);  
rclcpp::spin_until_future_complete(node, future);
```

```
auto resp = future.get();
```

Action Name

Goal Data

Initiate Action

Block Waiting





# Message vs. Service vs. Action



Type	Strengths	Weaknesses
Message	<ul style="list-style-type: none"><li>• Good for most sensors (streaming data)</li><li>• One - to - Many</li></ul>	<ul style="list-style-type: none"><li>• Messages can be <u>dropped</u> without knowledge</li><li>• Easy to overload system with too many messages</li></ul>
Service	<ul style="list-style-type: none"><li>• Knowledge of missed call</li><li>• Well-defined feedback</li></ul>	<ul style="list-style-type: none"><li>• Connection typically re-established for each service call (slows activity)</li></ul>
Action	<ul style="list-style-type: none"><li>• Monitor long-running processes</li><li>• Handshaking (knowledge of missed connection)</li></ul>	<ul style="list-style-type: none"><li>• Complicated</li></ul>





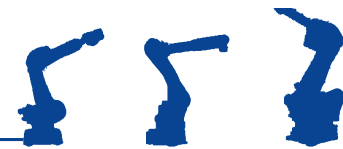
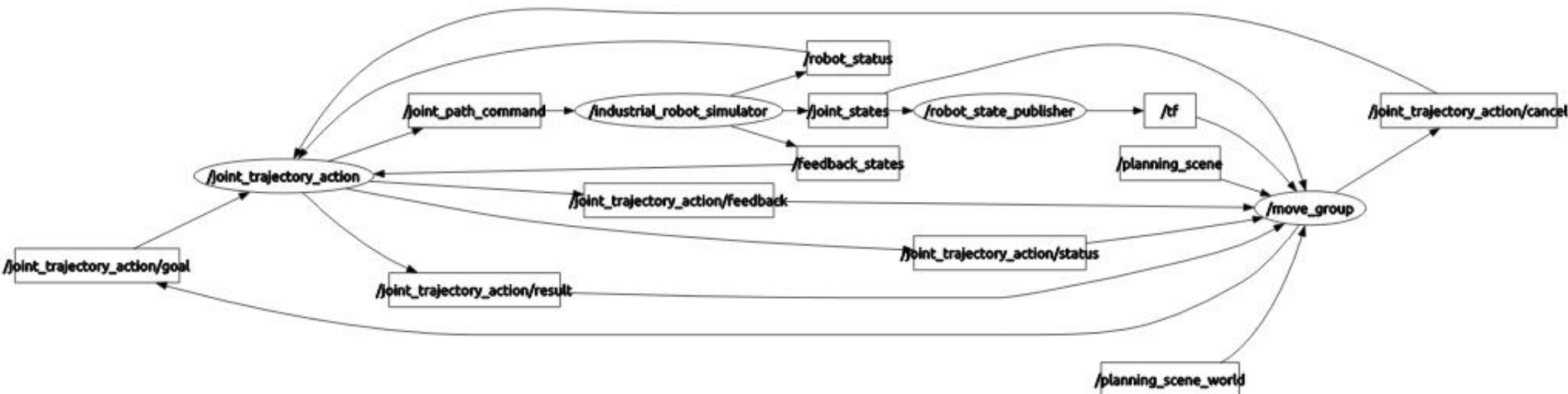
# Launch Files





# Launch Files: Motivation

- ROS is a Distributed System
  - often 10s of nodes, plus configuration data
  - painful to start each node “manually”



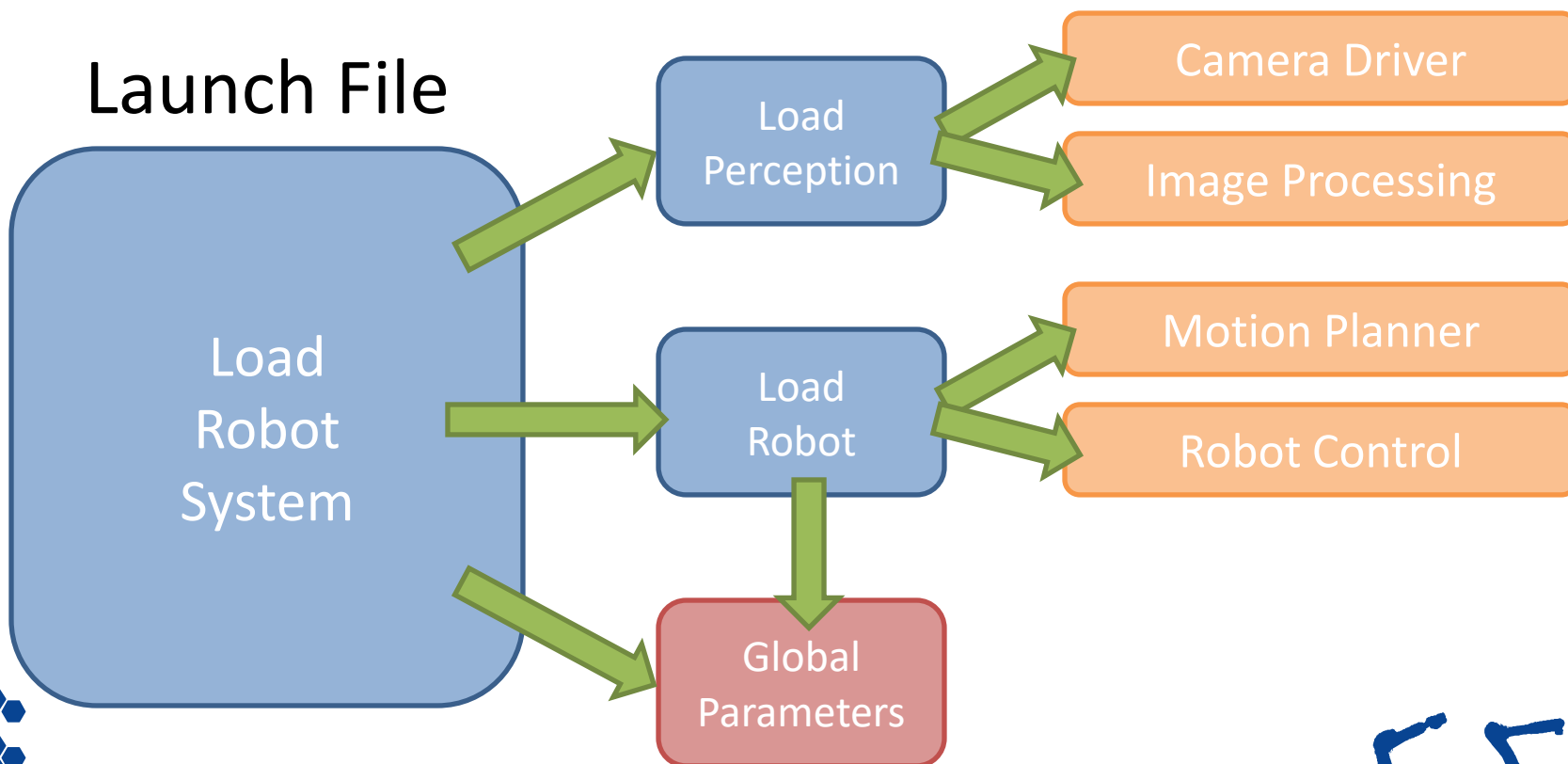


# Launch Files: Overview

Launch Files are like **Startup Scripts**

## Nodes

### Launch File







# Launch Files: Overview



- Launch files automate system startup
- Python script for running nodes and setting parameters
  - Python preferred, but XML and YAML also supported
  - ROS1 Launch files are typically XML
- Ability to pull information from other packages
  - load parameter (or other) data
  - “include” other launch files





- Script returns a list of launch **actions**
  - *Can use other Python logic to generate complex startup logic*

example.launch.py

```
import launch

def generate_launch_description():
    return launch.LaunchDescription(
        [
            <Action1>,
            <Action2>,
            ...
        ]
    )
```





## Common Actions

- **Node** - *launch a Node*
- **IncludeLaunchDescription** - *include other launch file*
- **DeclareLaunchArgument** - *define input arg*
- **GroupAction** - *define group of actions (e.g. for conditional)*
- **TimerAction** - *trigger action after a fixed period of time*





# Launch Files: Node Action



## `launch_ros.actions.Node (`

- **executable** — name of the executable file [REQUIRED]
- **package** — name of the package containing the executable
- **name** — unique name to assign to this node
- **namespace** — ROS namespace for this node
- **parameters** — node parameters to set (list of dictionaries or YAML filenames)
- **output** — control whether node output is echoed to the terminal window or not

```
launch_ros.actions.Node(  
    package = "usb_camera",  
    executable = "camnode",  
    name = "camera_1",  
    parameters = [{ 'ip_addr', "192.168.1.1" }],  
    output = 'screen',  
)
```





# Launch Files: Include



## `launch.actions.IncludeLaunchDescription()`

- `<1st arg>` – absolute filename of the launch file to include [REQUIRED]
- `launch_arguments` – dictionary of launch-file arguments

```
launch.actions.IncludeLaunchDescription(  
    PythonLaunchDescriptionSource(  
        get_package_share_directory('turtlesim') + '/launch/multisim.launch.py'  
    ),  
    launch_arguments={}.items()  
)
```





# Launch Files: Arguments



## `launch.actions.DeclareLaunchArgument` (

- `<1st arg>` – name of the input argument [REQUIRED]
- `default_value` – default value if no argument specified (makes this an OPTIONAL arg)
- `description` – user-friendly description of this argument

```
launch.actions.DeclareLaunchArgument(  
    'ip_addr',  
    default_value='192.168.1.1',  
    description='IP address of the robot'  
)  
  
launch_ros.actions.Node(  
    package = "abb_driver",  
    executable = "abb_robot_state",  
    parameters = [{ 'ip_addr', LaunchConfiguration('ip_addr') }],  
)
```





# Launch Files: Advanced



## Advanced features

- **remappings** – topic/service name remapping (list of (“old”, “new”) tuples)
- **condition** – conditional expression for whether to launch this node or not
- **GroupAction** – define group of actions
- **TimerAction** – delay actions by a specified period

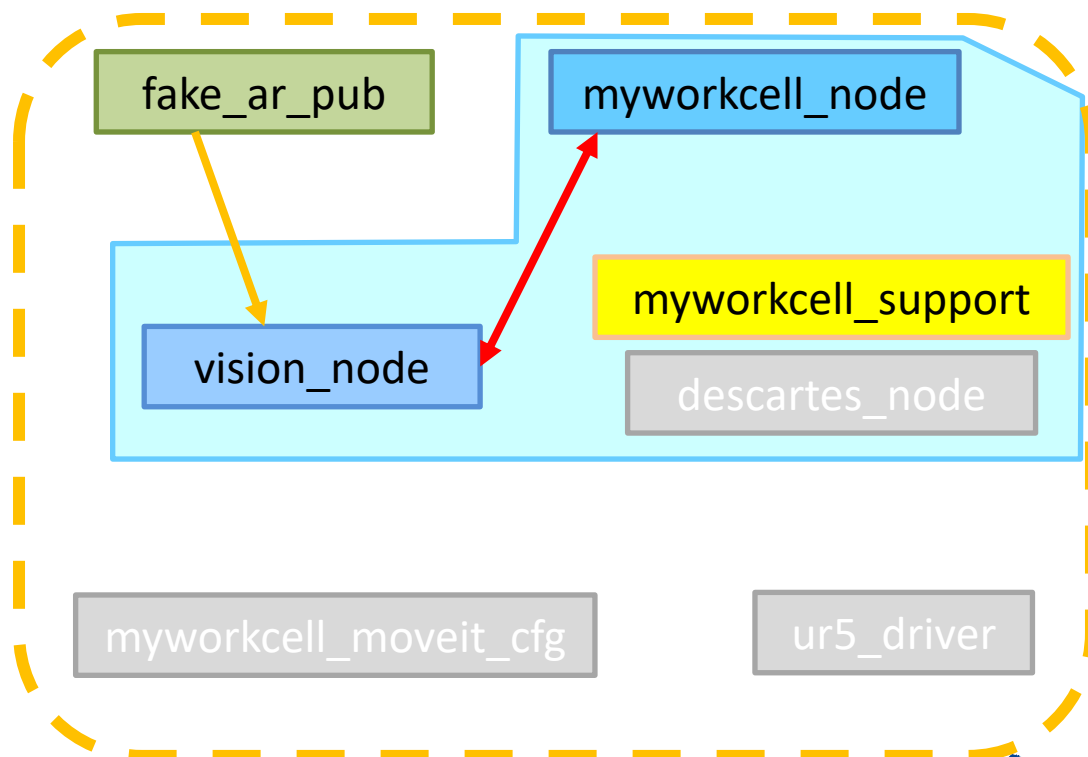
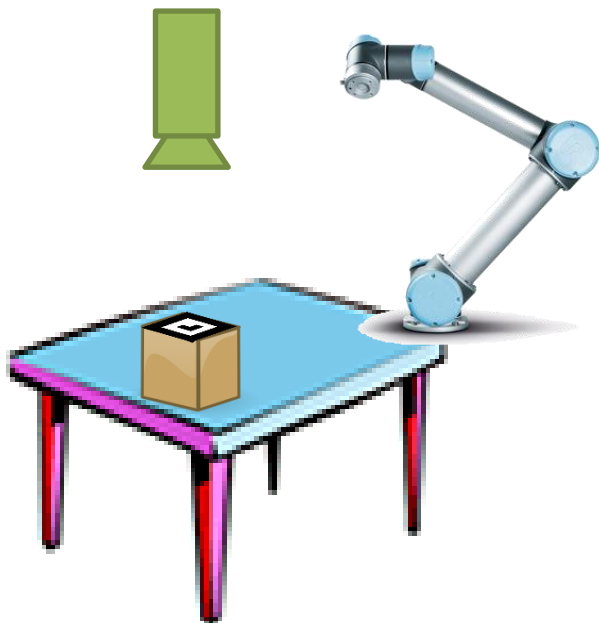
```
launch_ros.actions.Node(  
    ...  
    remappings = [('rgb', 'image')],  
)  
  
launch.actions.GroupAction(  
    [  
        Node(name='node1', ...),  
        Node(name='node2', ...),  
    ],  
    condition = IfCondition(use_robot)  
)  
  
launch.actions.TimerAction(  
    period=1.0,  
    actions=[  
        Node(name='imageProcessing', ...)  
    ]  
)
```





# Exercise 2.1

## Exercise 2.1 - Launch Files



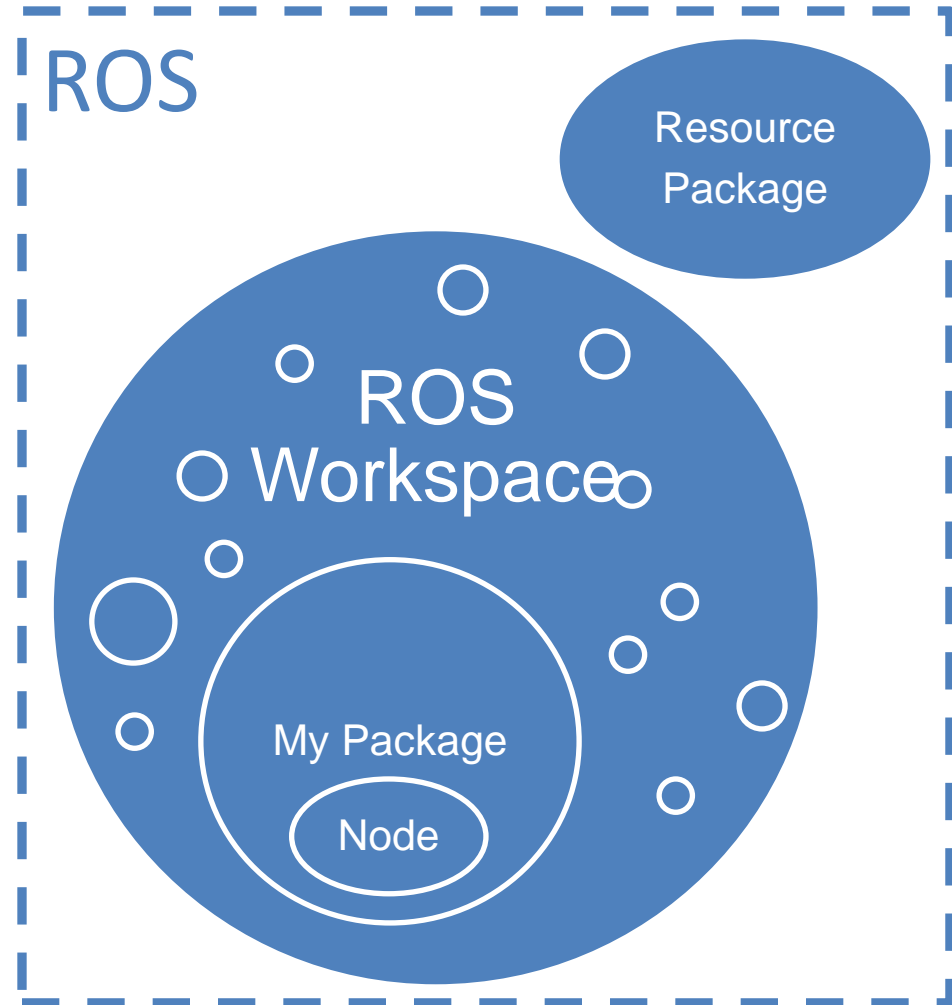




# Day 1 Progression



- ✓ Install ROS
- ✓ Create Workspace
- ✓ Add “resources”
- ✓ Create Package
- ✓ Create Node
  - ✓ Basic ROS Node
  - ✓ Interact with other nodes
    - ✓ Messages
    - ✓ Services
- ✓ Run Node
  - ✓ `ros2 run`
  - ✓ `ros2 launch`





# Parameters





# Parameters: Overview



Parameters are remotely-accessible **Global Data** associated with **each node**

Node1

```
debug  
ipAddr  
home_pos.x  
home_pos.y  
home_pos.z
```

Node2

```
debug  
ipAddr  
home_pos.x  
home_pos.y  
home_pos.z
```





# ROS Parameters



- Typically configuration-type values
  - robot kinematics
  - hardware config: IP Address, frame rate
  - algorithm limits / tuning
- Each Node manages its own parameters
  - can't get/set parameters when node is not running
- Parameter Notifications
  - typically parameters are loaded/read by nodes at startup
  - nodes can also register callbacks to get notified of parameter changes on-the-fly
  - this callback can also reject parameter changes, if invalid





# Parameter Datatypes



- Native Types
  - *int, real, boolean, string*
- Lists (vectors)
  - *of single type: [1.1, 1.2, 1.3]*
- Dictionaries (structures)
  - *translated to “dot” naming hierarchy in node*

```
box:  
  weight  
  center:  
    x  
    y
```



```
box.weight  
box.center.x  
box.center.y
```





# Setting Parameters



- YAML Files

```
manipulator_kinematics:  
  solver: kdl_plugin/KDLKinematics  
  search_resolution: 0.005  
  timeout: 0.005  
  attempts: 3
```

- Command Line

```
ros2 run my_pkg load_robot --ros-args -p ip:="192.168.1.21"  
ros2 param set load_robot /debug true
```

- Programs

```
node->set_parameter(rclcpp::Parameter("name", "left"));
```





# Parameter Commands



- **ros2 param**

- `ros2 param set <node> <key> <value>`
- `ros2 param get <node> <key>`
- `ros2 param delete <node> <key>`
- `ros2 param list <node>`
- `ros2 param dump <node>`
- `ros2 param load <node> <file.yaml>`





# Parameters: C++ API



- Accessed through `rclcpp::Node` object
  - `node->declare_parameter<type>(key, default)`  
*Declare parameter for this node (with default value)*
  - `node->get_parameter(key).as_int()`  
*Gets value. Must use helper method to convert to std type.*
  - `node->set_parameter(rclcpp::Parameter(<key>, <value>))`  
*Sets value. Need to construct the Parameter object.*
- This API requires you to explicitly declare parameters
  - no on-the-fly updating
  - typically read only when node first started







# Dynamic Parameters

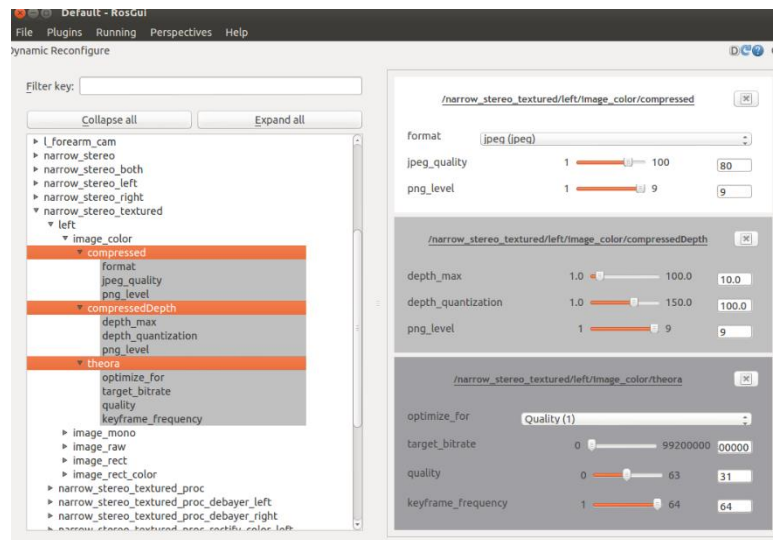


- For dynamic params: register a callback

```
SetParametersResult paramCB(const vector<Parameter> &params)
{
    // loop over changed params
    // react to those changes (save to local vars, push to h/w)
    // set result.successful to accept/reject changes
}

this->set_on_parameters_set_callback(&paramCB);
```

rqt\_reconfigure GUI

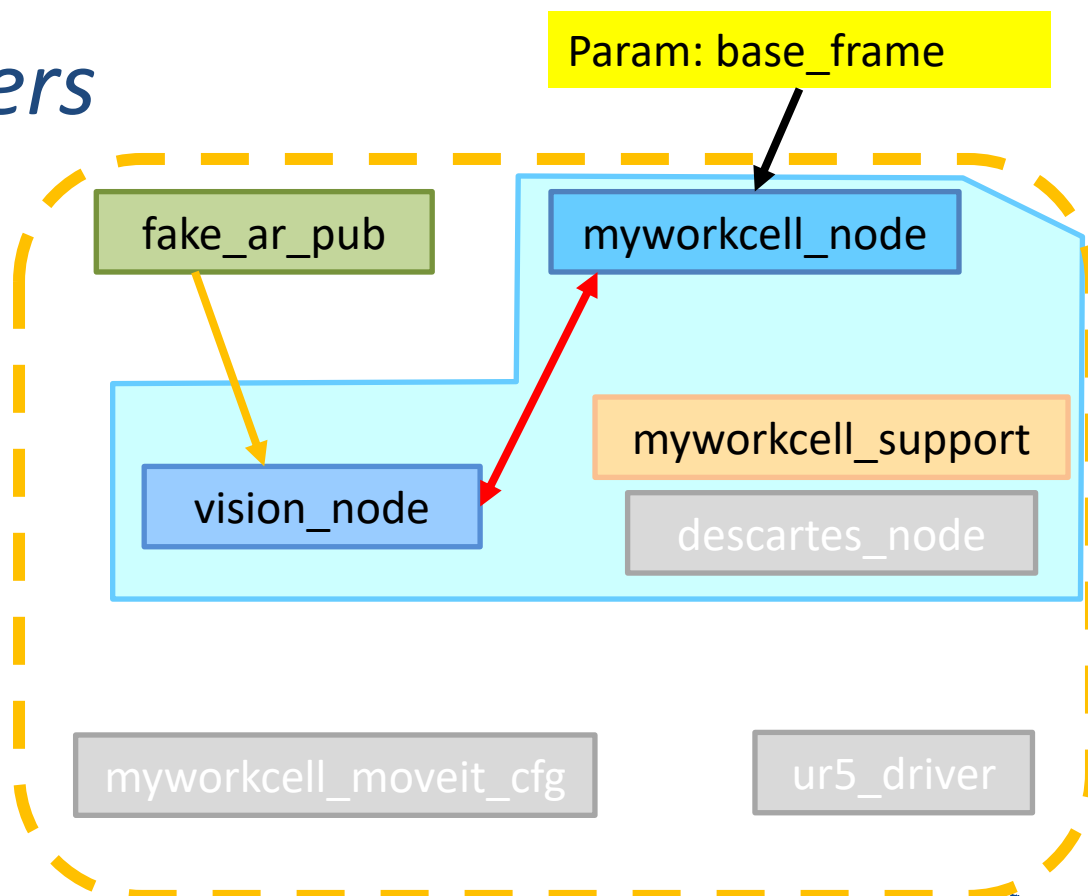
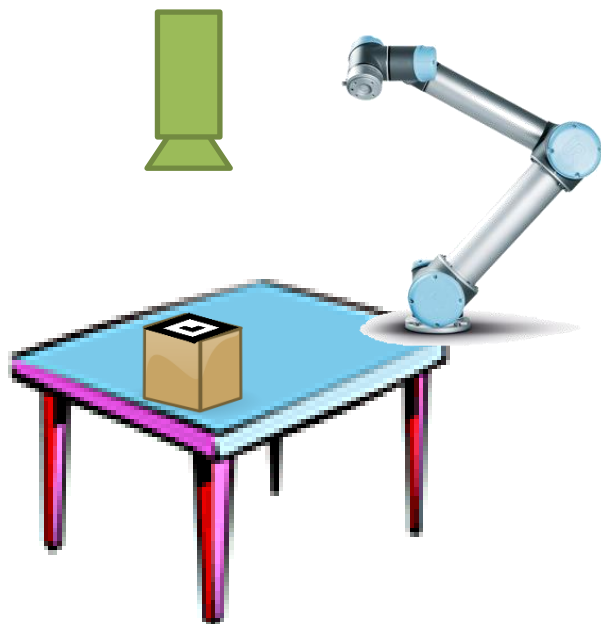




# Exercise 2.2

## Exercise 2.2

### *ROS Parameters*





# Review/Q&A



## Session 1

Intro to ROS

Installing ROS/Packages

Packages

Nodes

Messages/Topics

## Session 2

Services

Actions

Launch Files

Parameters

