Robotics Programming with ROS2

Lecture 2 : ROS2 Fundamentals

BY

PI THANACHA CHOOPOJCHAROEN

Agenda

- History of ROS
- Getting to know ROS
- Introduction to ROS 2
- Navigating Linux Terminal
- Using ROS2 in Terminal
- ROS2 Node Programming with RCLPY
- OOP with RCPLY
- Custom Interface, Service Server, Service Client
- Action, Multithread Execution, Callback Group

Main Objective:

To read the RCLPY-based code, create and customize basic ROS2 node.

Brief History of ROS

Before becoming ROS (Stanford University)

Two majors common problem in robotics community

Developers takes too much time developing basic structure of their robotics software.

Developers have little time working on new cutting-edge software for their robots.

https://www.youtube.com/watch?v=9J9kxb_7dUg

Founders: Keenan Wyrobek และ Eric Berger (2006)

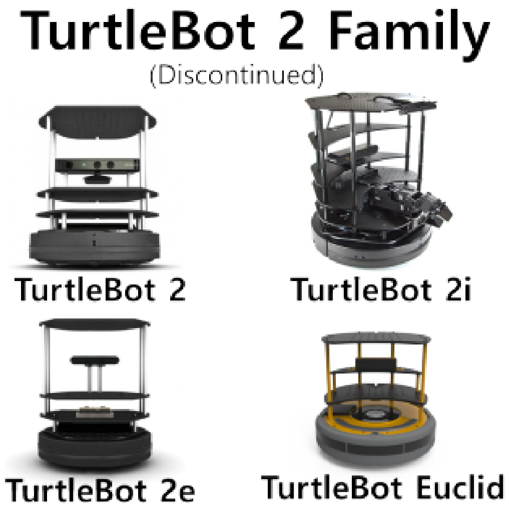
The birth of ROS at Willow Garage (Menlo Park California)

Scott Hassan (Founder & investor) was interested and invested in ROS.
He started a new department called "Personal Robotics Program" in 2008.

Products from Willow Garage

PR1

PR2



Open Source
Robotics Foundation

Willow Garage shutdown in 2014. It spun off to several companies. , one being “Open Robotics”, formerly known as "Open Source Robotics Foundation".

DARPA
ROBOTICS
CHALLENGE

https://www.youtube.com/watch?v=g0TaYhjpOfo

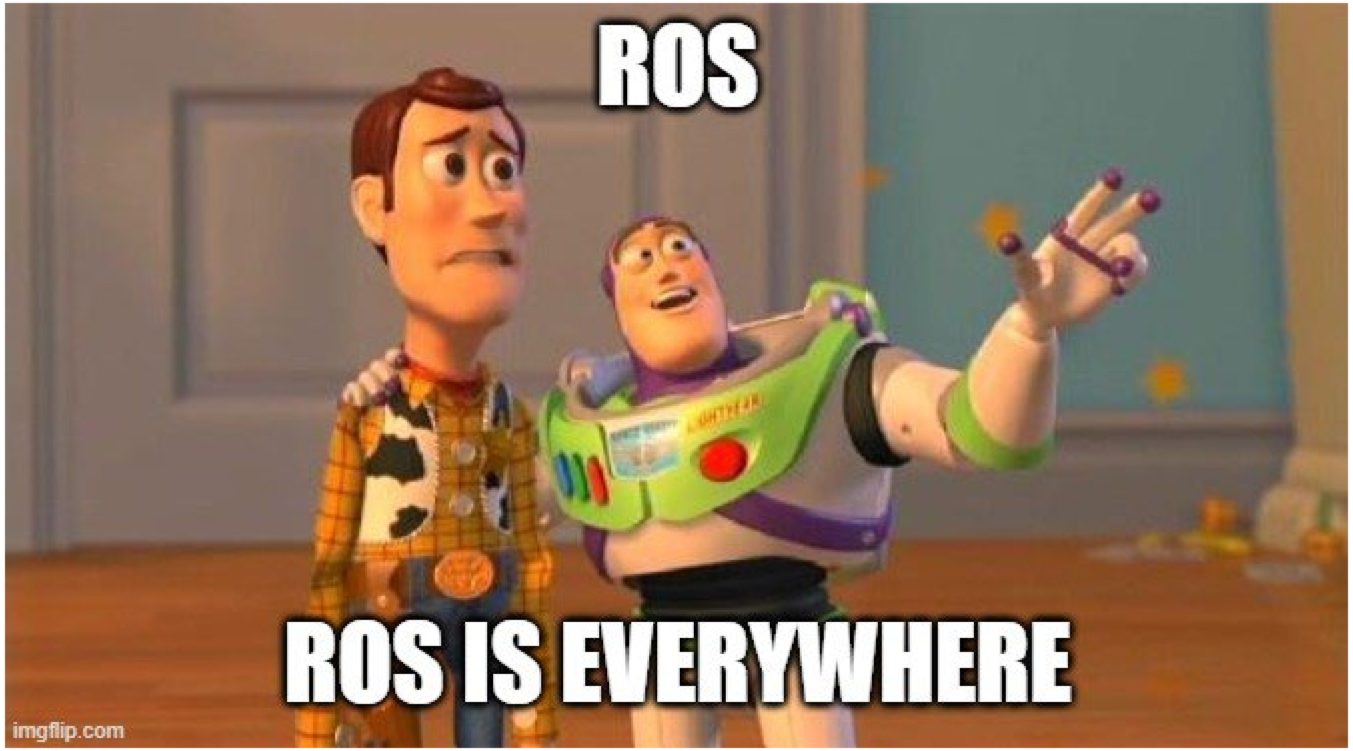
DARPA
SUBTERRANEAN
CHALLENGE

https://www.youtube.com/watch?v=HuJGIAjuxLE

open robotics

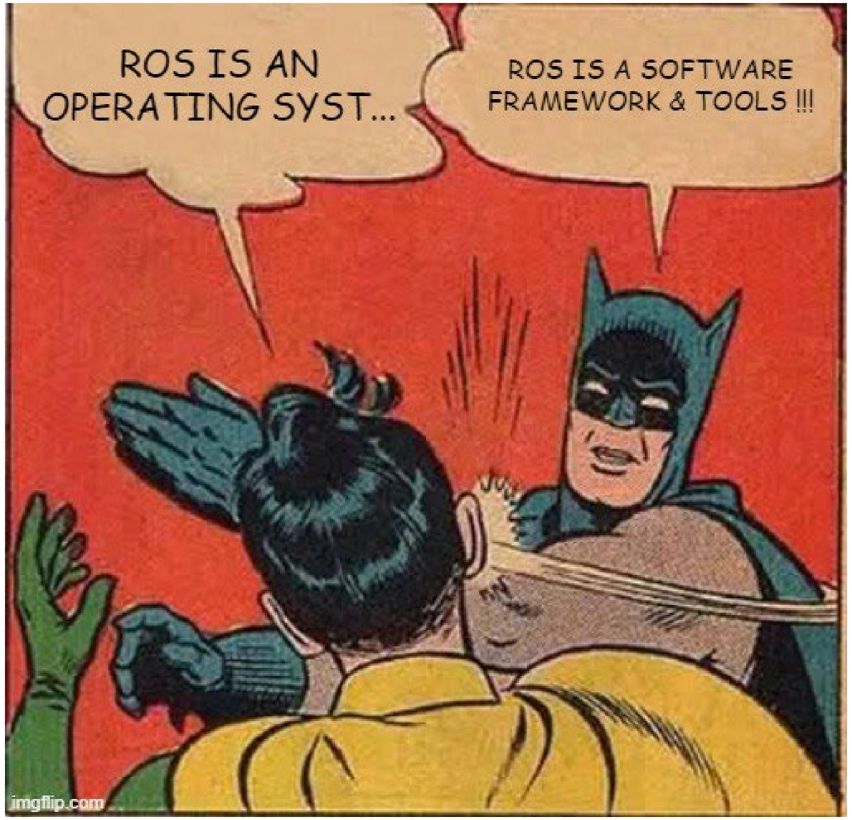
Open Robotics is a non-profit corporation that is responsible for ROS, Gazebo Simulator, Ignition, and RMF.

In 2018, the company opened a new headquater in SIngapore. Open Robotics also collaborate with the goverment of Singapore to become a pioneer in Robotics medical tech.



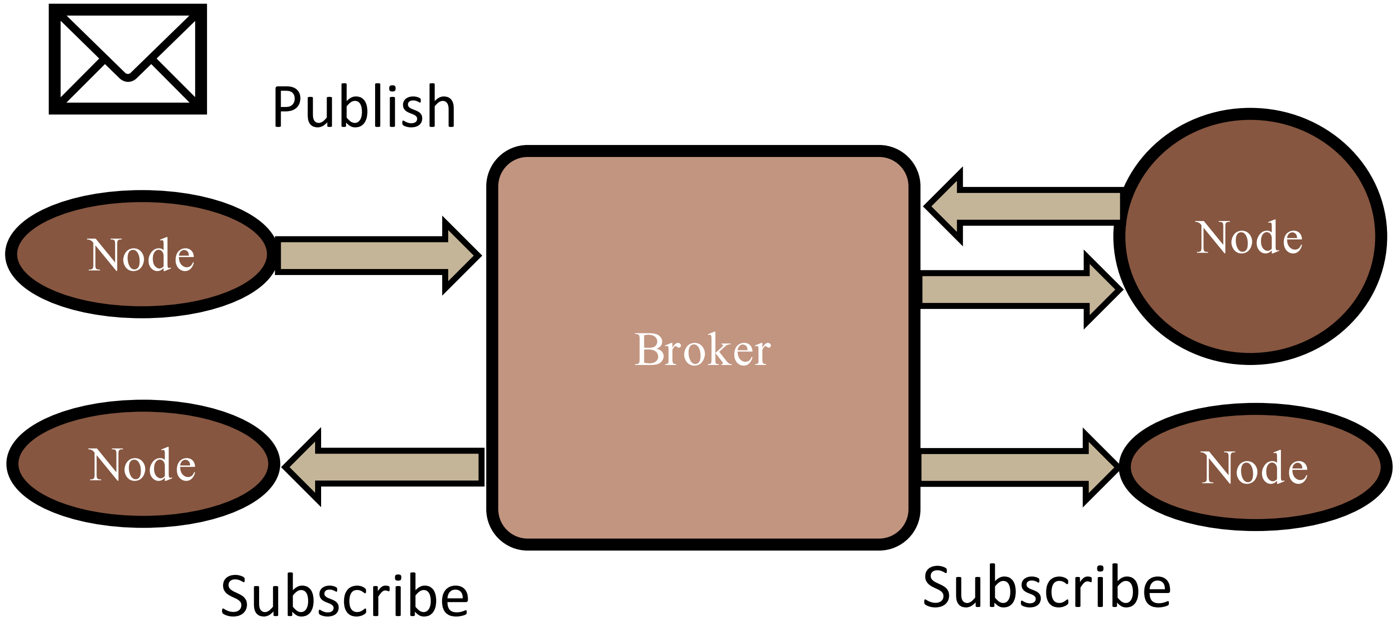
Getting to know ROS

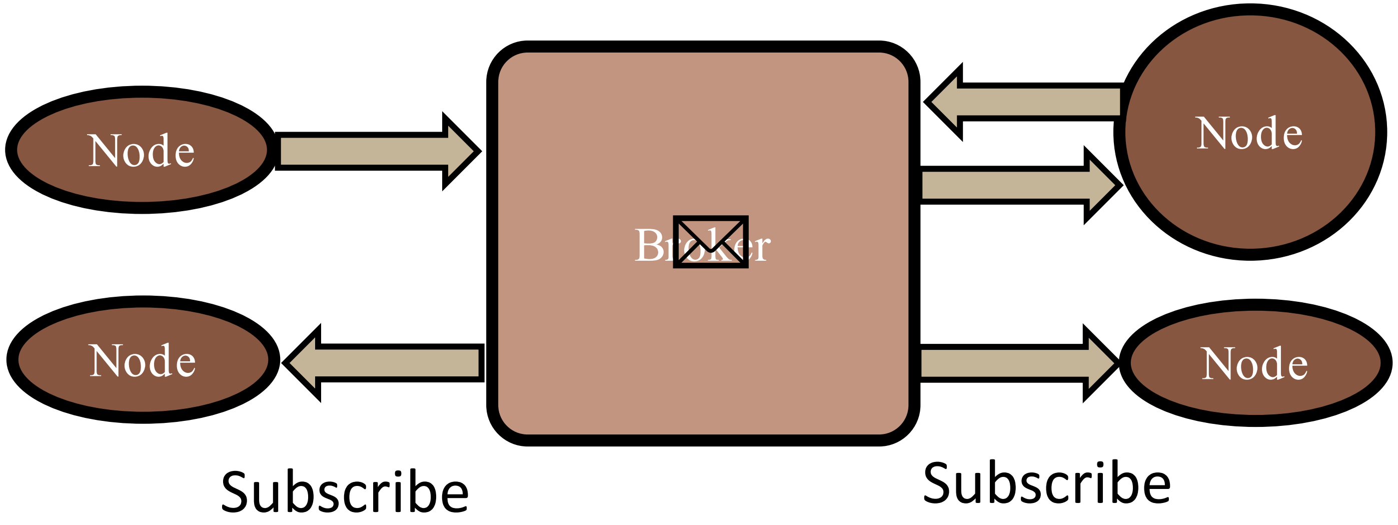
What is/ isn't ROS ?

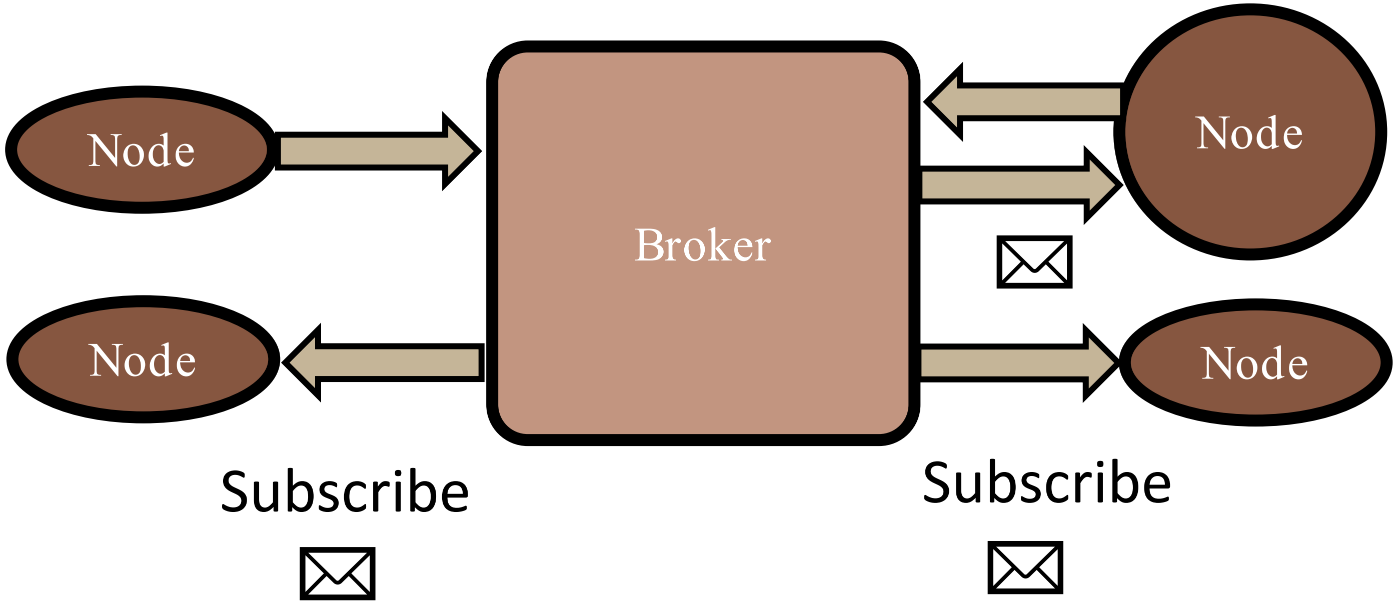

ROS IS AN OPERATING SYST...
ROS IS A SOFTWARE FRAMEWORK & TOOLS !!!
imgflip.com

Robot Operating System (ROS) was often mistaken as an operating system due to its misleading name. ROS is actual a software framework for developing systems that consists of a cluster of computers. It is often used with a robotics hardware that have operating system.

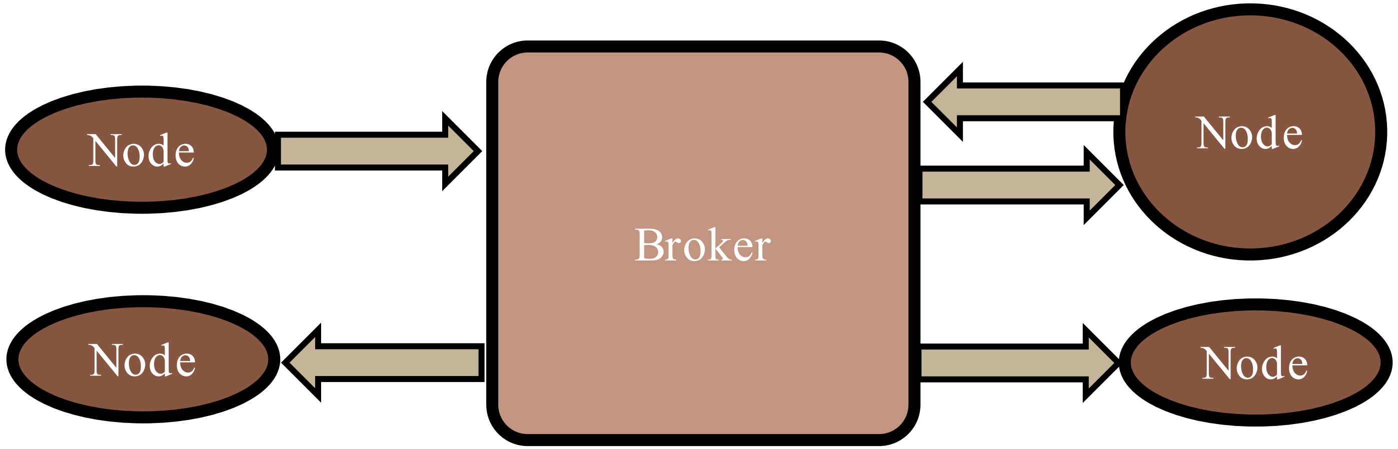
ROS


Publish
Node
Broker
Node
Node
Node
Subscribe
Subscribe


Node
Node
Broker
Node
Node
Subscribe
Subscribe


Node
Node
Broker
Node
Node
Subscribe
Subscribe

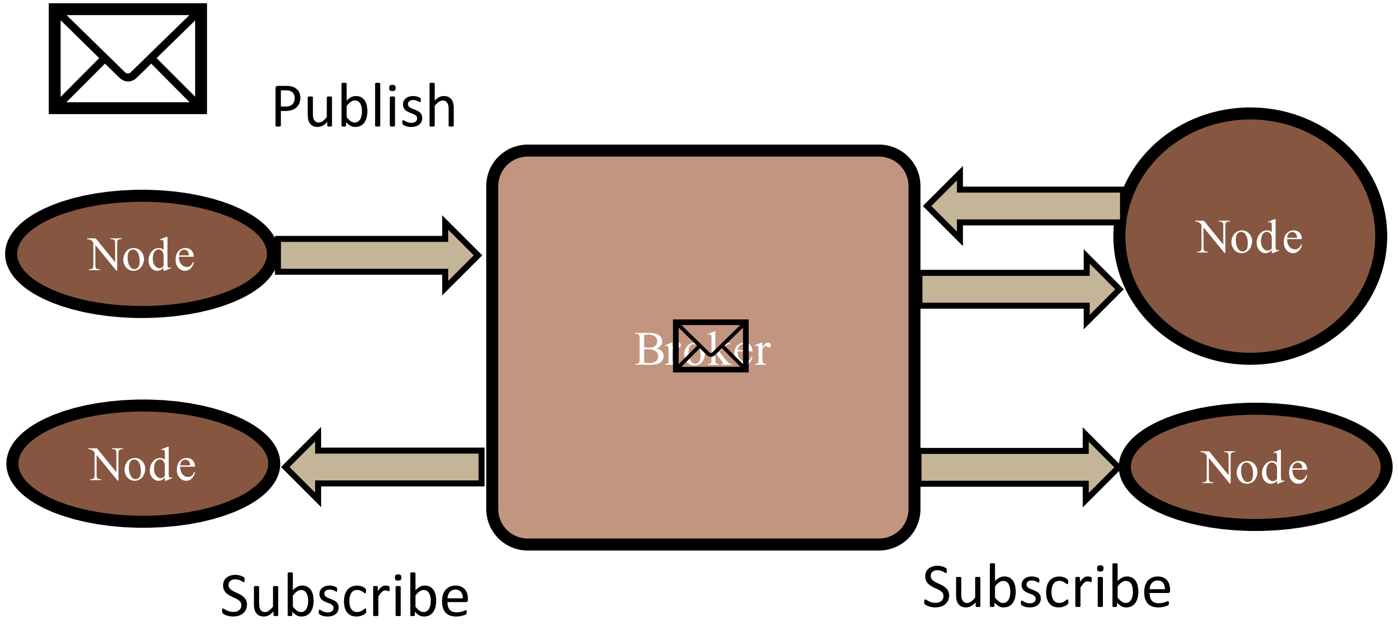
ROS

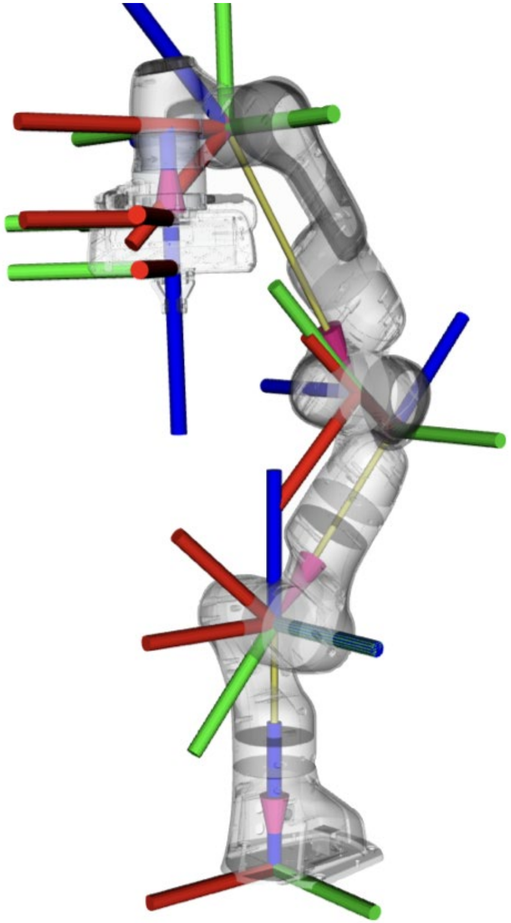


MQTT

OPC UA

kafka


Publish
Node
Broker
Node
Node
Node
Subscribe
Subscribe



What does ROS do ????

Tools, Standards, Conventions

Resource Sharing Platform

Suported Programming Language for ROS

Suported Programming Language for ROS

MATLAB®

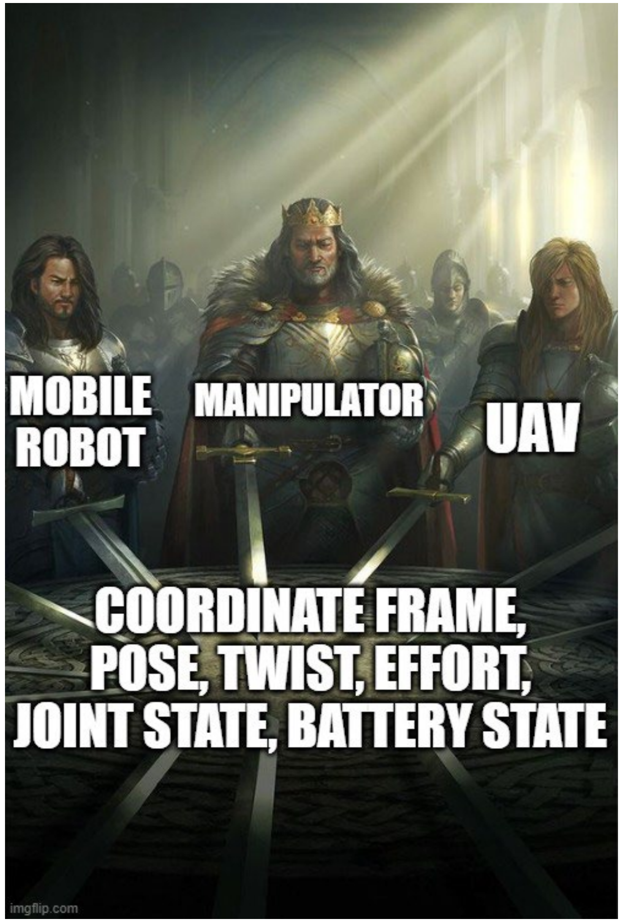
Java

Lua

The Rust Programming Language

julia

Common type of information used in robotics


MOBILE ROBOT
MANIPULATOR
UAV
COORDINATE FRAME, POSE, TWIST, EFFORT, JOINT STATE, BATTERY STATE
imgflip.com

ROS Message Types

Accel
Inertia
Point
Point32
Pose
Pose2D
Quaternion
Transform
Twist
Vector3
Wrench

BatteryState
CameraInfo
CompressedImage
Image
Imu
JointState
Joy
LaserScan
PointCloud
Temperature

Packages for/by the ROS Community

If your idea is simple, it is most likely to exist as a ROS package.

rviz

gazebo

aruco_detect

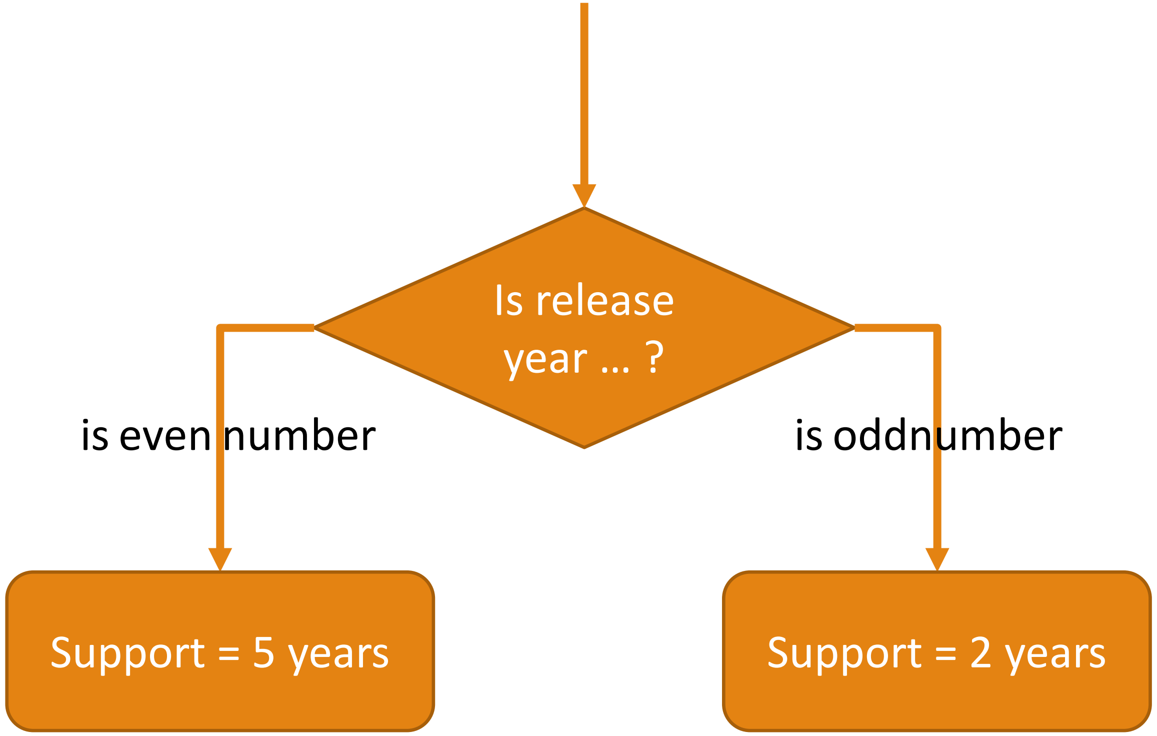
global_planner

ros_control

map_server

camera_calibration

ROS Distribution & End-of-Life (EOL)


Is release year … ?

is even number

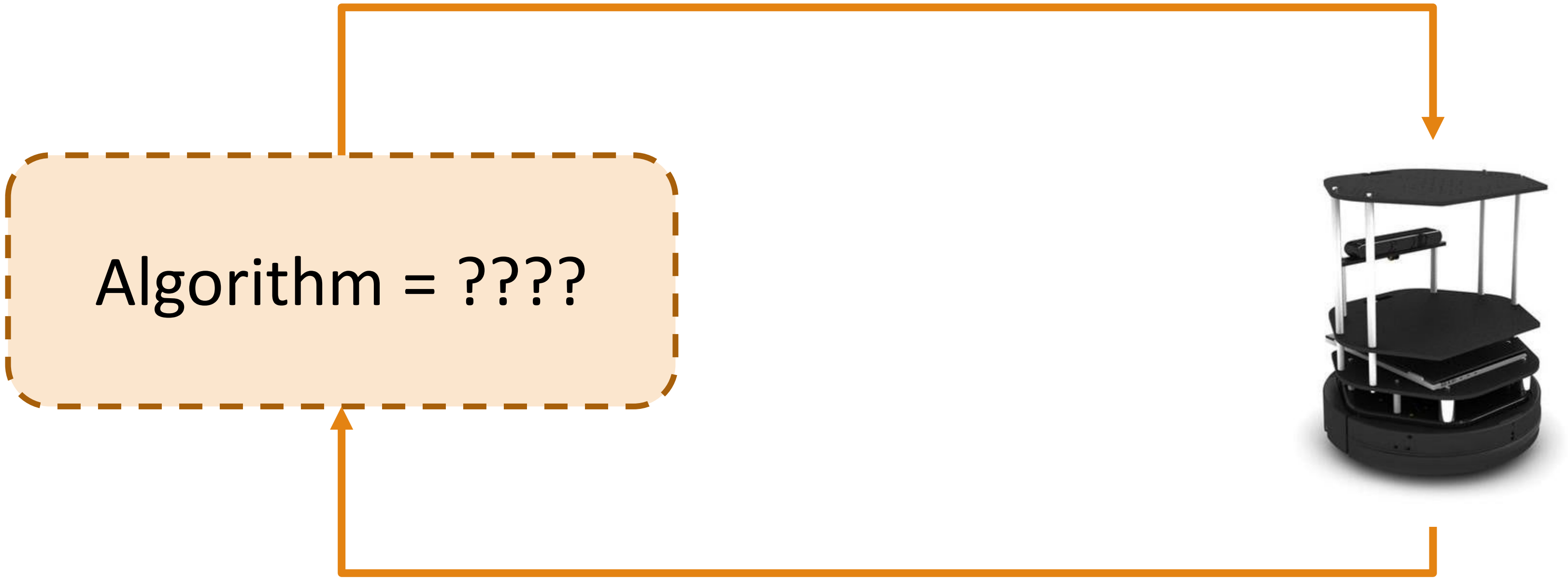
is oddnumber

Support = 5 years

Support = 2 years

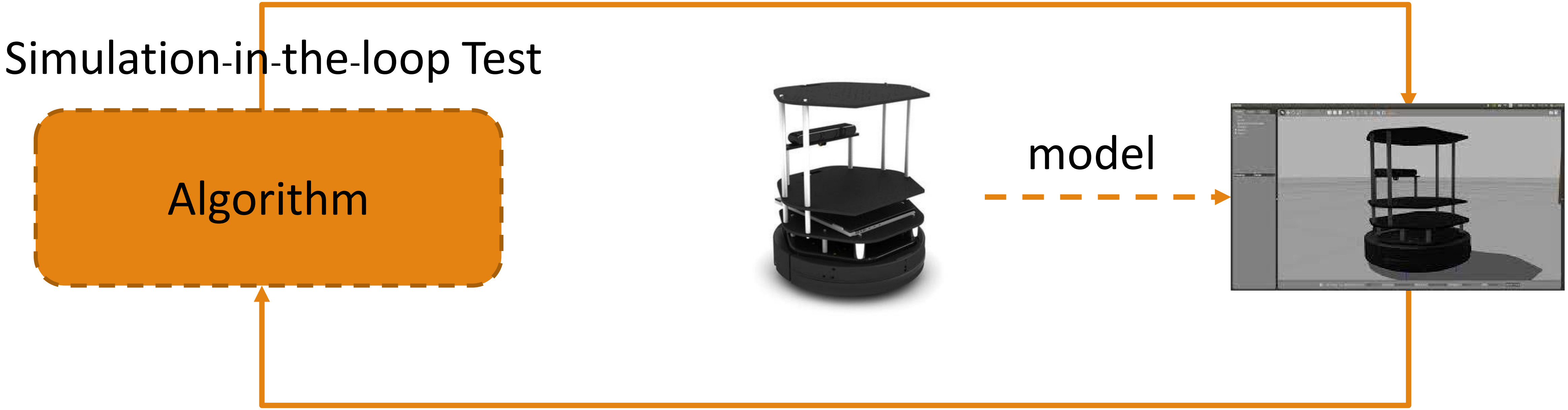
Distro	Release date	EOL date
ROS Noetic Ninjemys	May 23rd, 2020	May, 2025
ROS Melodic Morenia	May 23rd, 2018	May, 2023
ROS Lunar Loggerhead	May 23rd, 2017	May, 2019
ROS Kinetic Kame	May 23rd, 2016	April, 2021
ROS Jade Turtle	May 23rd, 2015	May, 2017
ROS Indigo Igloo	July 22nd, 2014	April, 2019
ROS Hydro Medusa	September 4th, 2013	May, 2015
ROS Groovy Galapagos	December 31, 2012	July, 2014
ROS Fuerte Turtle	April 23, 2012	--
ROS Electric Emys	August 30, 2011	--
ROS Diamondback	March 2, 2011	--
ROS C Turtle	August 2, 2010	--
ROS Box Turtle	March 2, 2010	--

Physical System vs Simulation



Algorithm = ????

Physical System vs Simulation

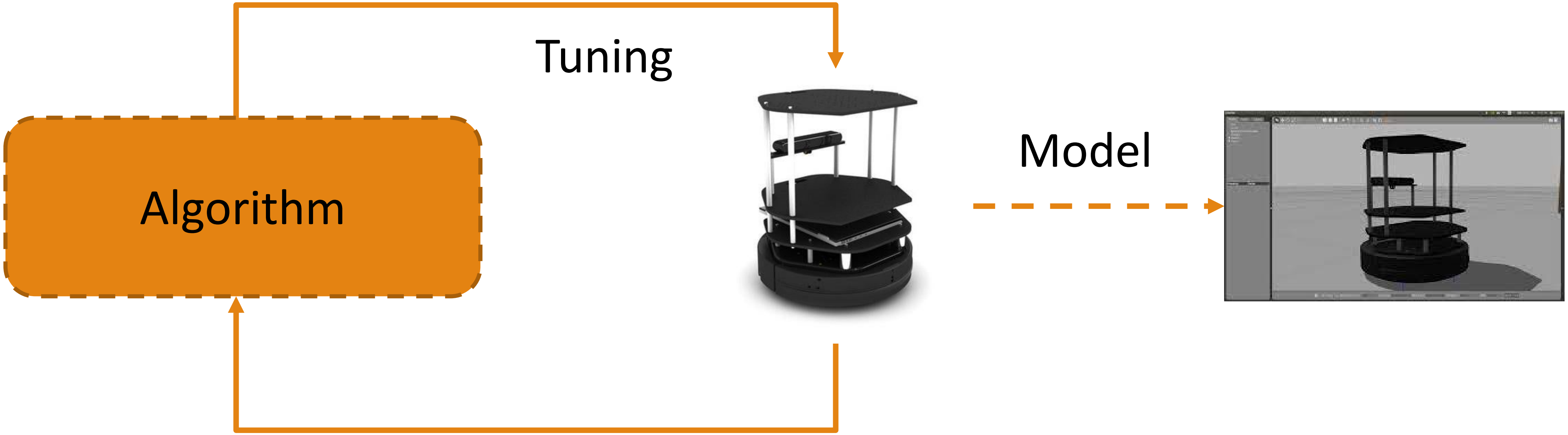


Simulation-in-the-loop Test

Algorithm

model

Physical System vs Simulation



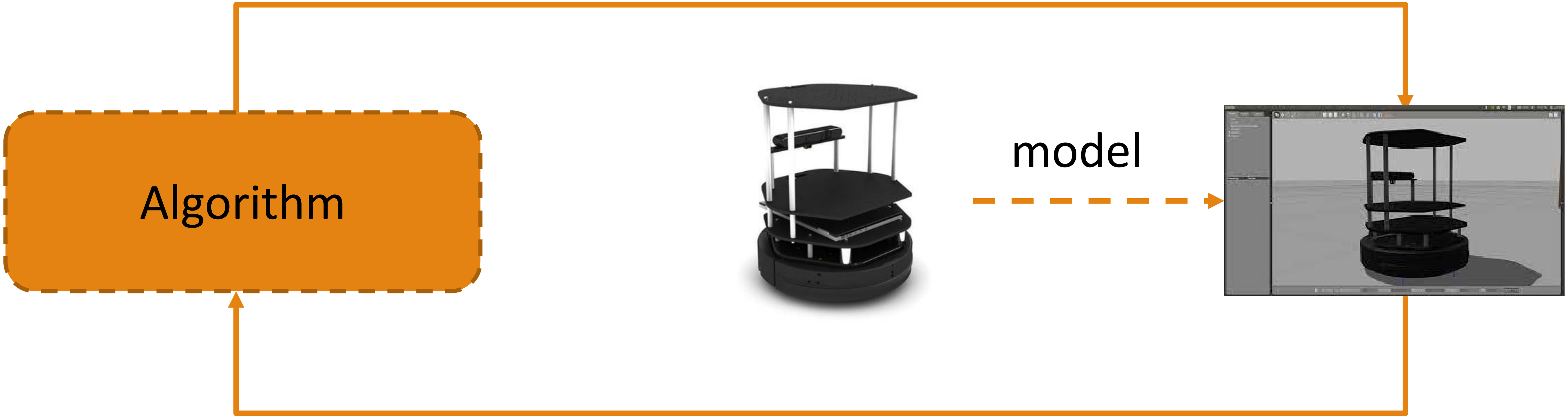
Tuning

Algorithm

Model

Physical System vs Simulation

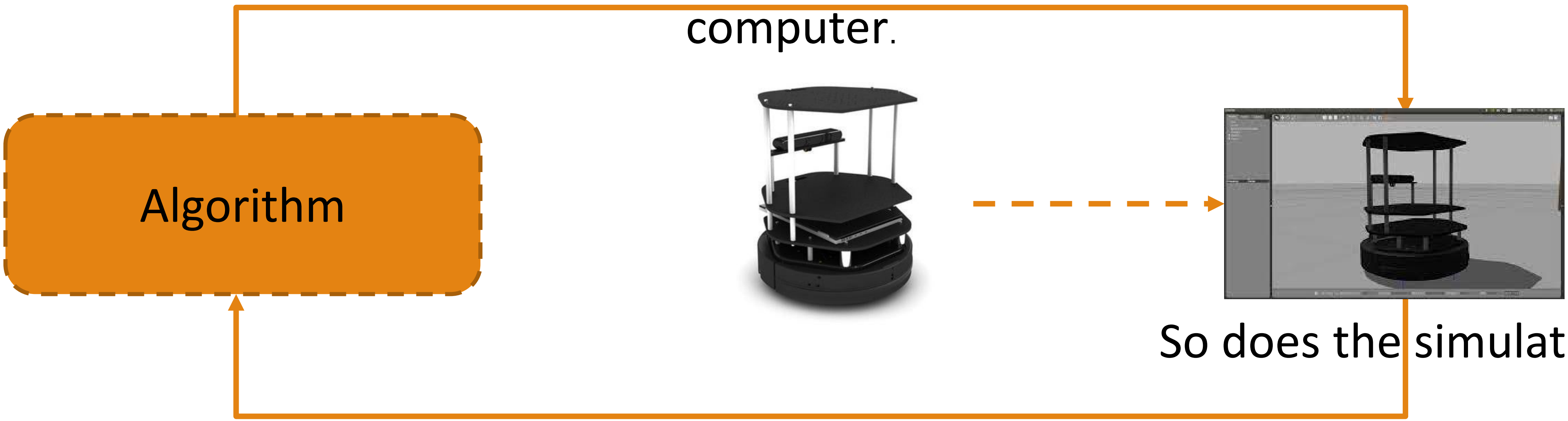
Tuning with the simulation



Algorithm

model

Physical System vs Simulation

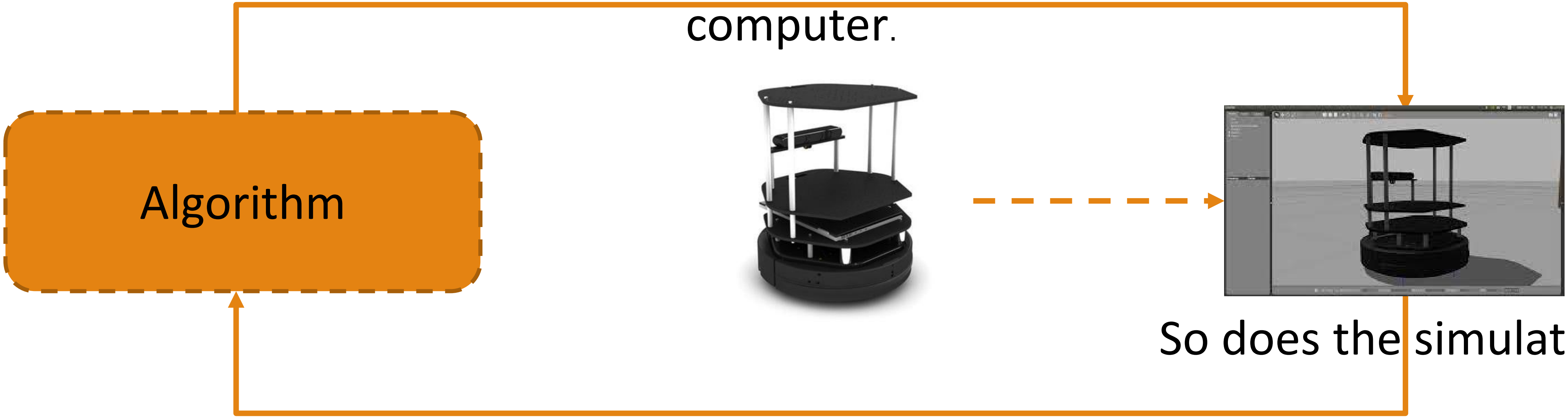


A robot needs a computer.

Algorithm

So does the simulation.

Physical System vs Simulation

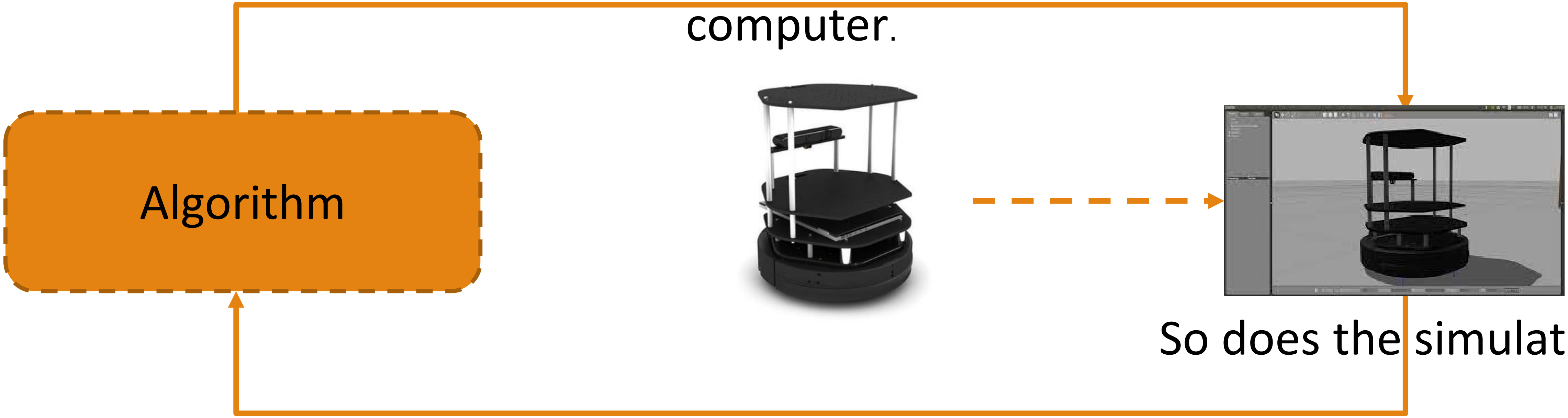


A robot needs a computer.

So does the simulation.

Q1.) Should they use the same computer ?

Physical System vs Simulation

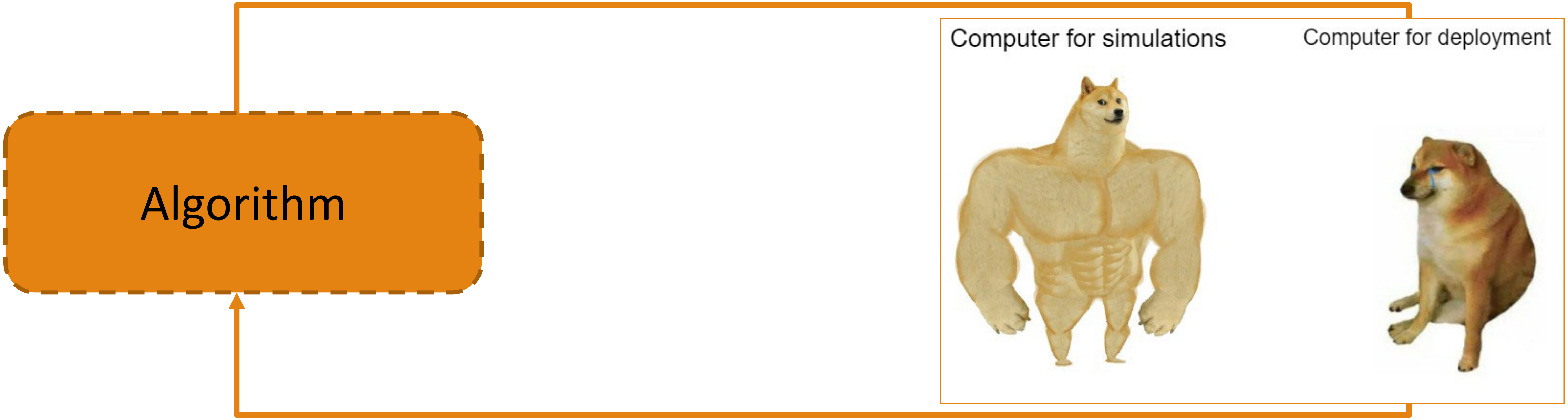


A robot needs a computer.

So does the simulation.

Q2.) Which one should be more powerful ?

Physical System vs Simulation



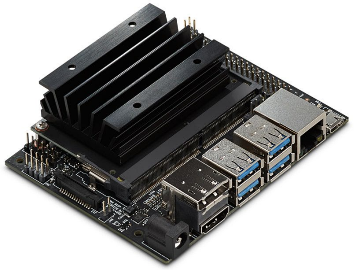
System Requirement for ROS

Hardware

- Graphic Card for simulator
- >1.6 GHz, Dual-core (Intel i5 processor or better)
- 4 GB RAM

Operating System

- Ubuntu, Fedora
- ***Windows 10,11 (some bug still exists in old distro)



Introduction to ROS2

Issues from using ROS (1)

ROS was not designed to support a system of multiple robots due to its used of a single master node.

ROs was designed so that the resources should be local (contains within the computer) for optimal performance

The paradigm of communication was not design to support realtime operation.

ROS was essentially a centralized system, which is not designed to support fleet management.

ROS 2

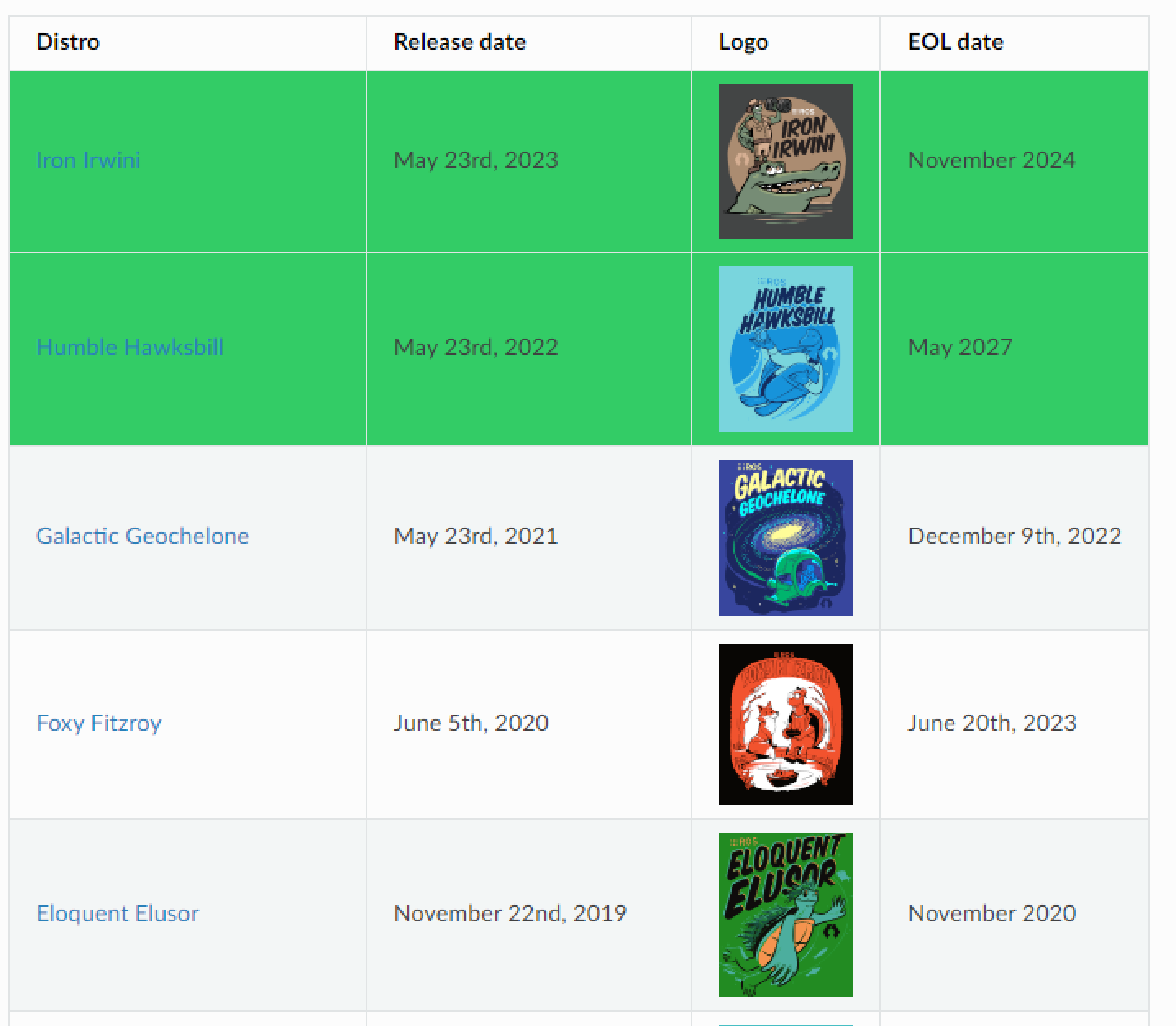
Some of many benefits of ROS2

Utilize Data Distribution System instead of TCPROS

Support multiple operating system such as linux, Windows, and OSX

Launch files can be writen using Python script !!!

ROS2 Distro



Distro	Release date	Logo	EOL date
Iron Irwini	May 23rd, 2023		November 2024
Humble Hawksbill	May 23rd, 2022		May 2027
Galactic Geochelone	May 23rd, 2021		December 9th, 2022
Foxy Fitzroy	June 5th, 2020		June 20th, 2023
Eloquent Elusor	November 22nd, 2019		November 2020

ROS 2 : Humble
(Ubuntu 22.04)

Let's use ROS2 !!!

Navigating Linux Terminal

Linux: File System & Path

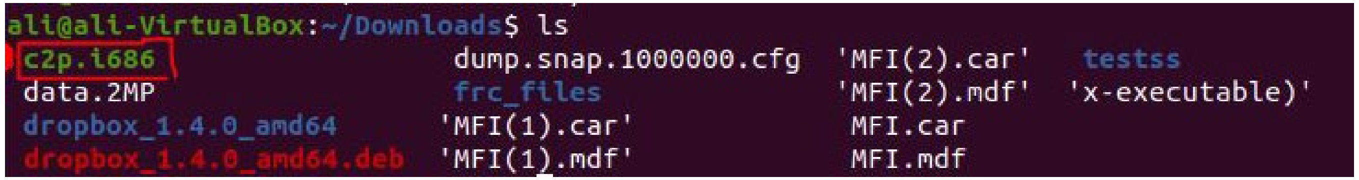
"Directory" : where one can keep their files or other sub-directories.

"File" : a content that requires an extension. (.jpg, .py, .cpp)

"Data file" : contains data that can be read or written by an editor or other programs.

"Executable file" : can be executed to perform certain task(s)

"Path" : describe the location of a directory or file



```
ali@ali-VirtualBox:~/Downloads$ ls
 c2p.i686                  dump.snap.1000000.cfg   'MFI(2).car'   testss
 data.2MP                  frc_files               'MFI(2).mdf'  'x-executable)'
 dropbox_1.4.0_amd64      'MFI(1).car'              MFI.car
 dropbox_1.4.0_amd64.deb  'MFI(1).mdf'              MFI.mdf
```

Common questions one should ask themselves

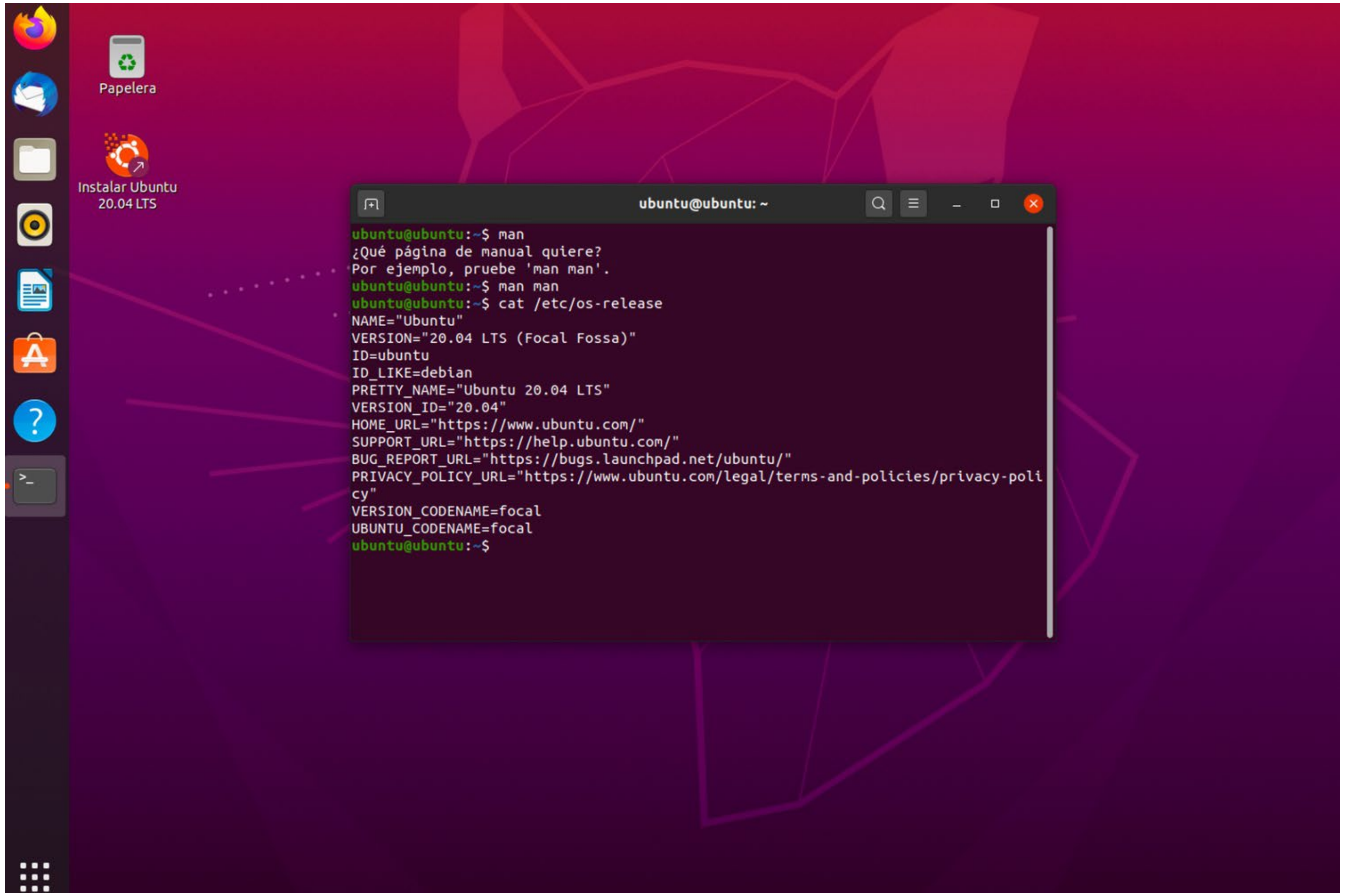
Which files does our system need ?

How can we make sure that the system will use the correct files regardless of the computer it was installed on ?

What should the system behave when the required file does not exist ?

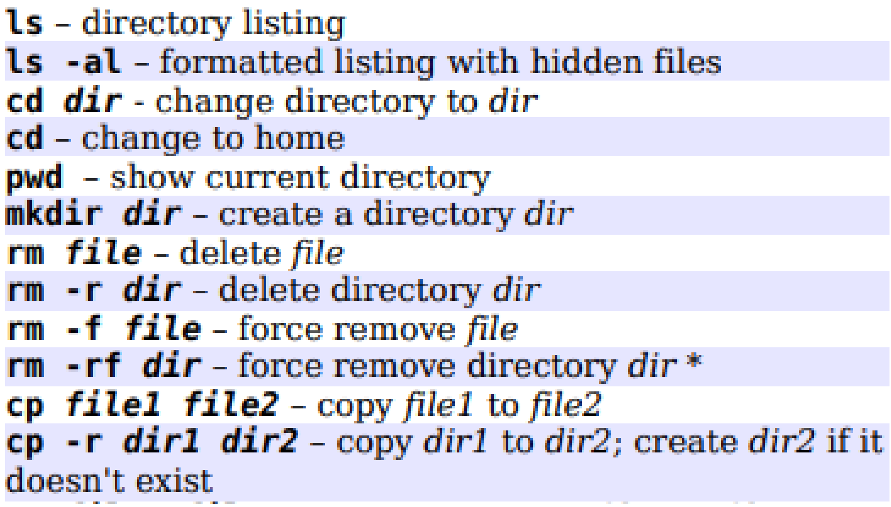
Ubuntu : Terminal & Command

One can execute a command in a terminal.



```
ubuntu@ubuntu:~$ man
¿Qué página de manual quiere?
Por ejemplo, pruebe 'man man'.
ubuntu@ubuntu:~$ man man
ubuntu@ubuntu:~$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04 LTS"
VERSION_ID="20.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-poli
cy"
VERSION_CODENAME=focal
UBUNTU_CODENAME=focal
ubuntu@ubuntu:~$
```

Ubuntu : Common Command



ls – directory listing

ls -al – formatted listing with hidden files

cd *dir* - change directory to *dir*

cd – change to home

pwd – show current directory

mkdir *dir* – create a directory *dir*

rm *file* – delete *file*

rm -r *dir* – delete directory *dir*

rm -f *file* – force remove *file*

rm -rf *dir* – force remove directory *dir* *

cp *file1 file2* – copy *file1* to *file2*

cp -r *dir1 dir2* – copy *dir1* to *dir2*; create *dir2* if it doesn't exist

List all installed ROS package

>> apt list ros-humble* --installed

Install a ROS package

>> sudo apt install ros-humble-[PACKAGE]

Open file explorer from the current working direcrtory

>> nautilus .

Modify bashrc

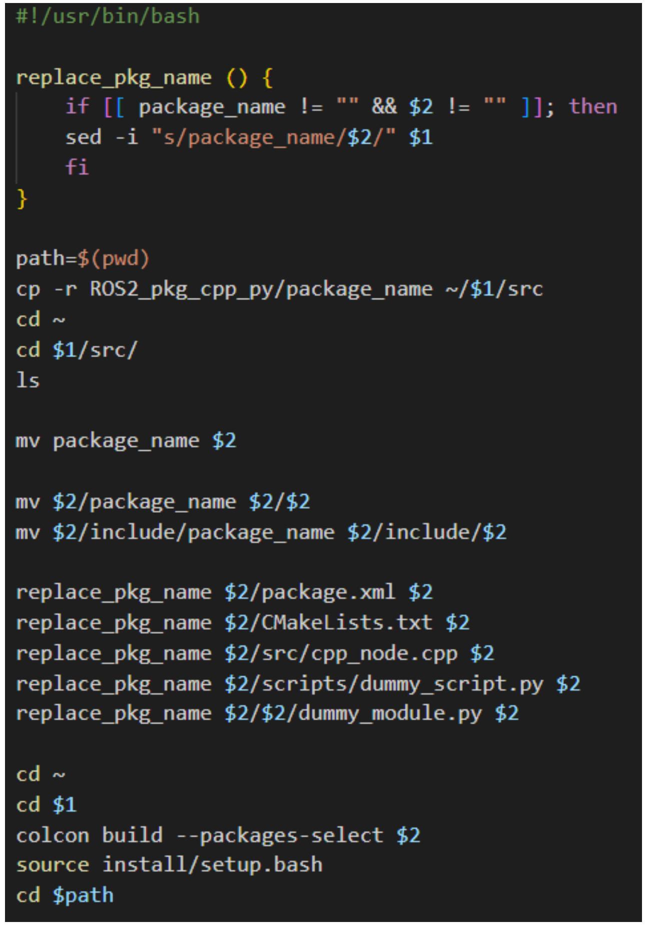
>> code ~/.bashrc

Bash Programming

```
>> cd
>> code my_script.bash
```

```
if [ -d "$1" ];
then
    cd $1
else
    echo "The directory $1 does not exist."
    echo "Creating directory $1"
    mkdir -p $1/src
    cd $1
fi
```

```
>> source my_script.bash
```



```
#!/usr/bin/bash

replace_pkg_name () {
    if [[ package_name != "" && $2 != "" ]]; then
    sed -i "s/package_name/$2/" $1
    fi
}

path=$(pwd)
cp -r ROS2_pkg_cpp_py/package_name ~/$1/src
cd ~
cd $1/src/
ls

mv package_name $2

mv $2/package_name $2/$2
mv $2/include/package_name $2/include/$2

replace_pkg_name $2/package.xml $2
replace_pkg_name $2/CMakeLists.txt $2
replace_pkg_name $2/src/cpp_node.cpp $2
replace_pkg_name $2/scripts/dummy_script.py $2
replace_pkg_name $2/$2/dummy_module.py $2

cd ~
cd $1
colcon build --packages-select $2
source install/setup.bash
cd $path
```

.bashrc

.bashrc is a hidden bash script in Home directory that will be called automatically when opening a new terminal.

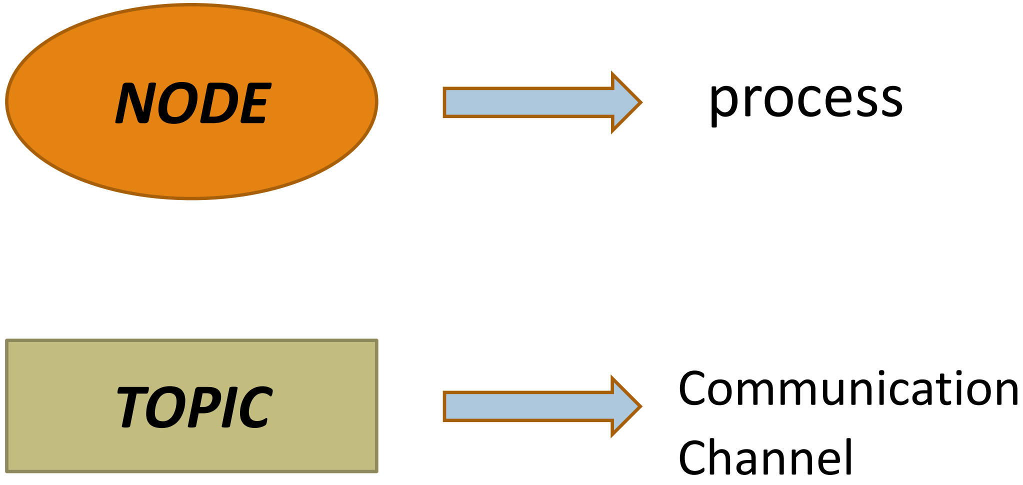
```
echo Please enter new terminal name :
read terminal_name
TITLE=“\[\e]0;$terminal_name\a\]”
PS1=$PS1${TITLE}
```

```
>> cd
>> code .bashrc
```

```
>> code ~/.bashrc
```

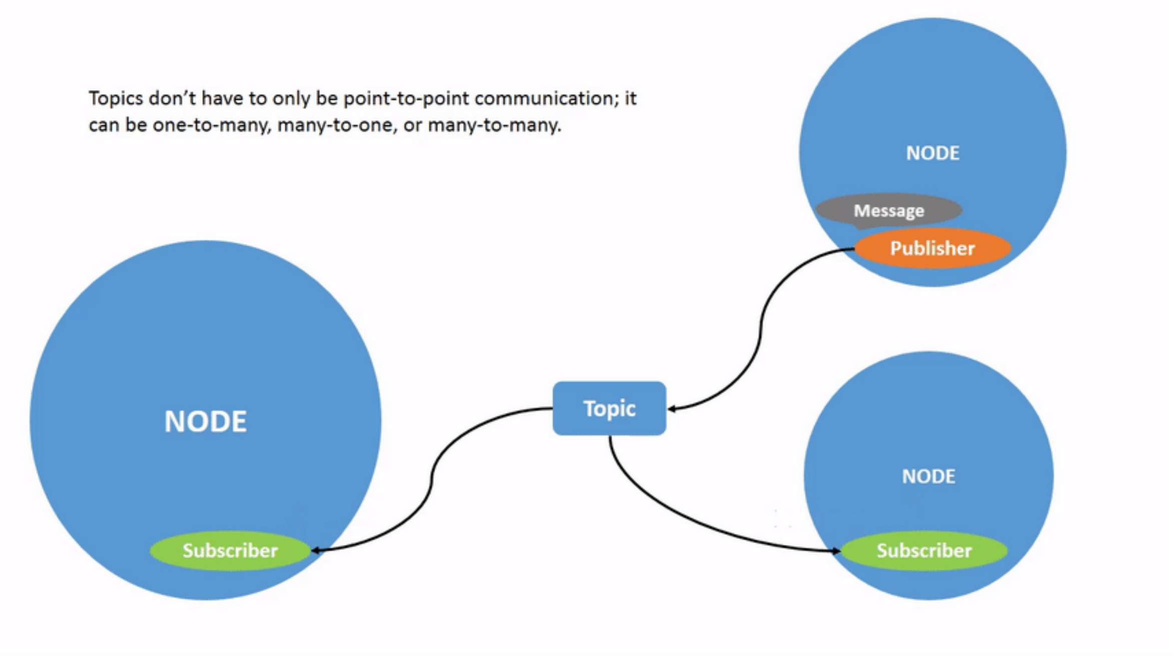
Using ROS2 in terminal

The concept of Nodes and Topics



NODE → process

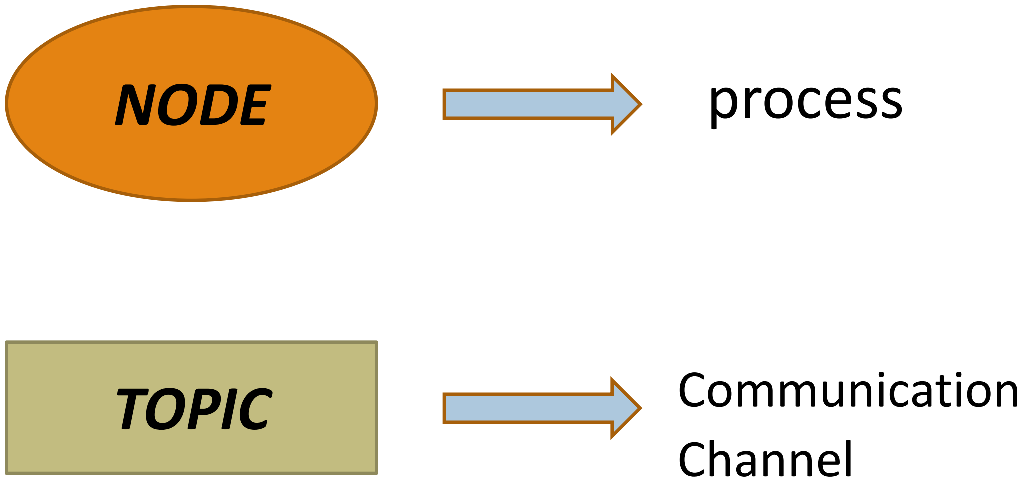
TOPIC → Communication Channel



Topics don’t have to only be point-to-point communication; it can be one-to-many, many-to-one, or many-to-many.

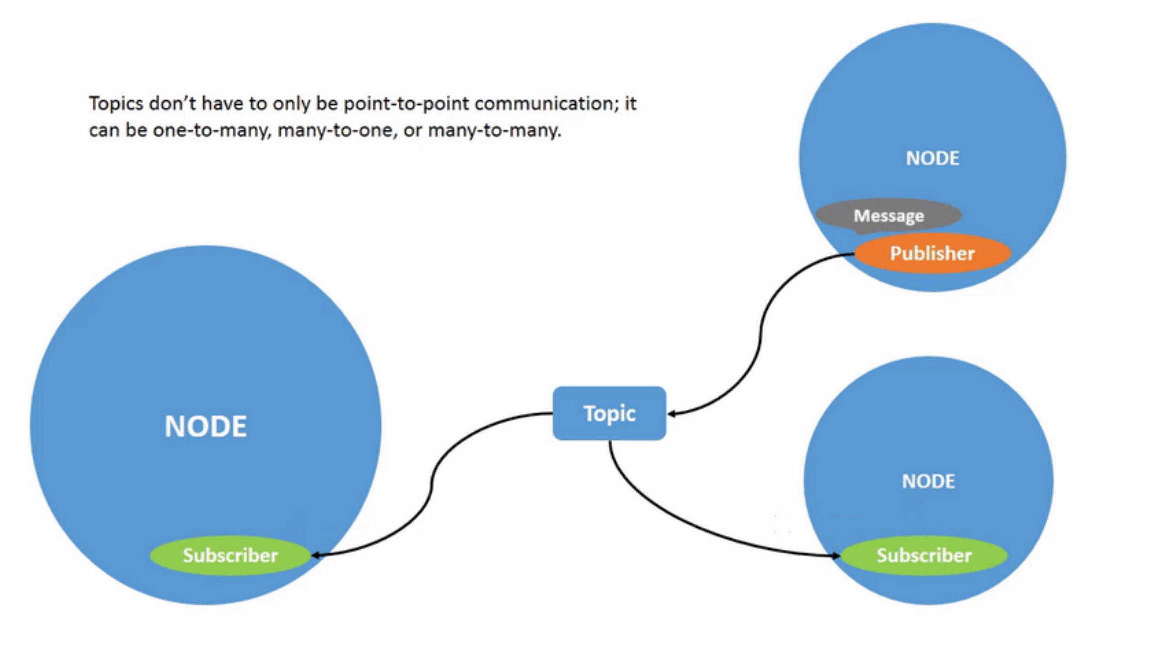
Node can have a publisher that publish a message to a topic.
Node can also have a subscription that subscribe a message from a topic.

The concept of Nodes and Topics



NODE → process

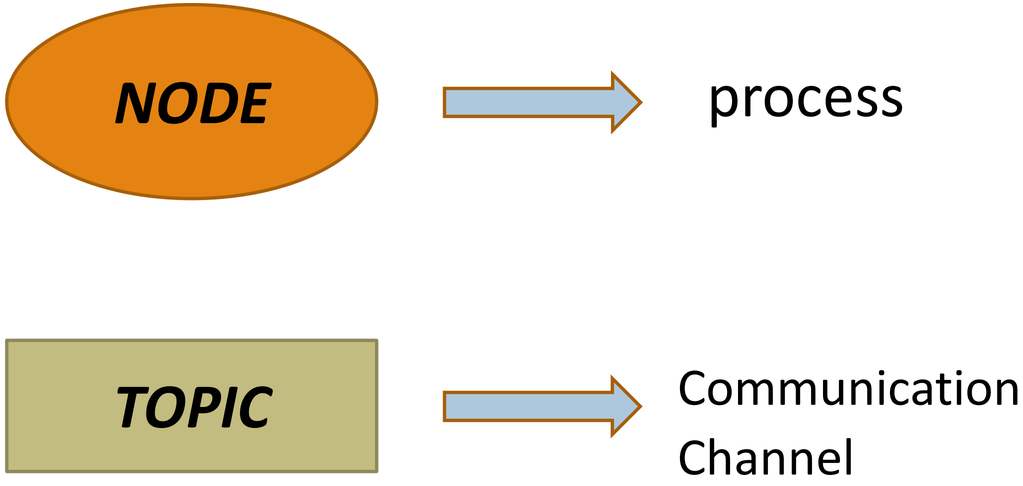
TOPIC → Communication Channel



Topics don't have to only be point-to-point communication; it can be one-to-many, many-to-one, or many-to-many.

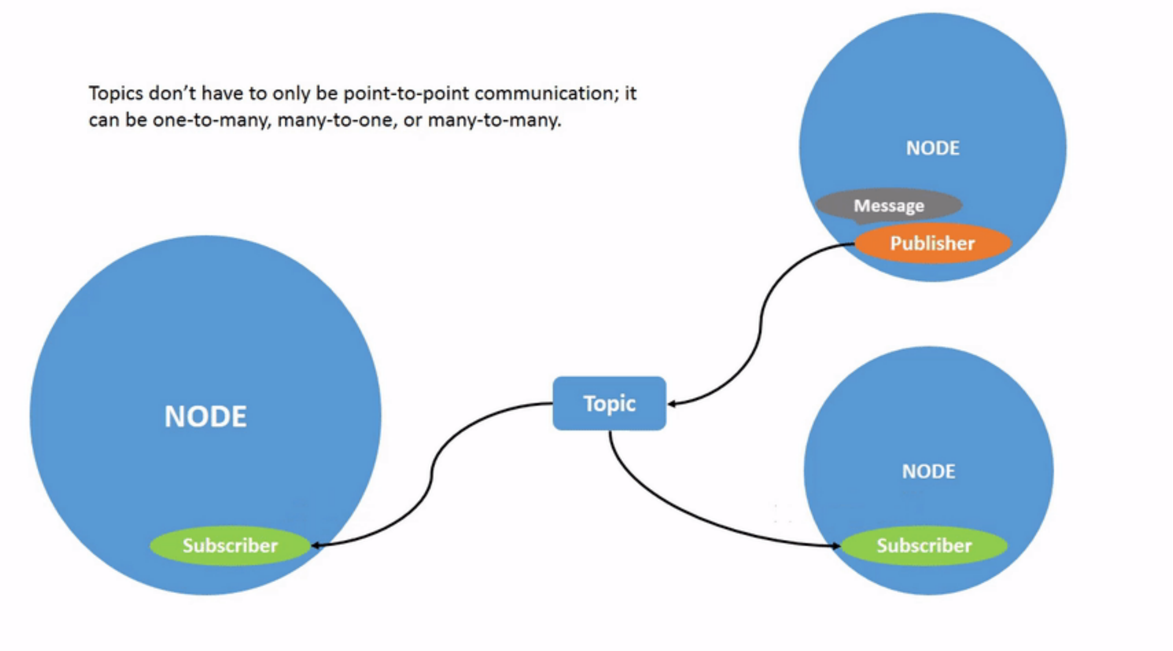
Q1.) Can a node have both a publisher and a subscription to the same topic ?

The concept of Nodes and Topics



NODE → process

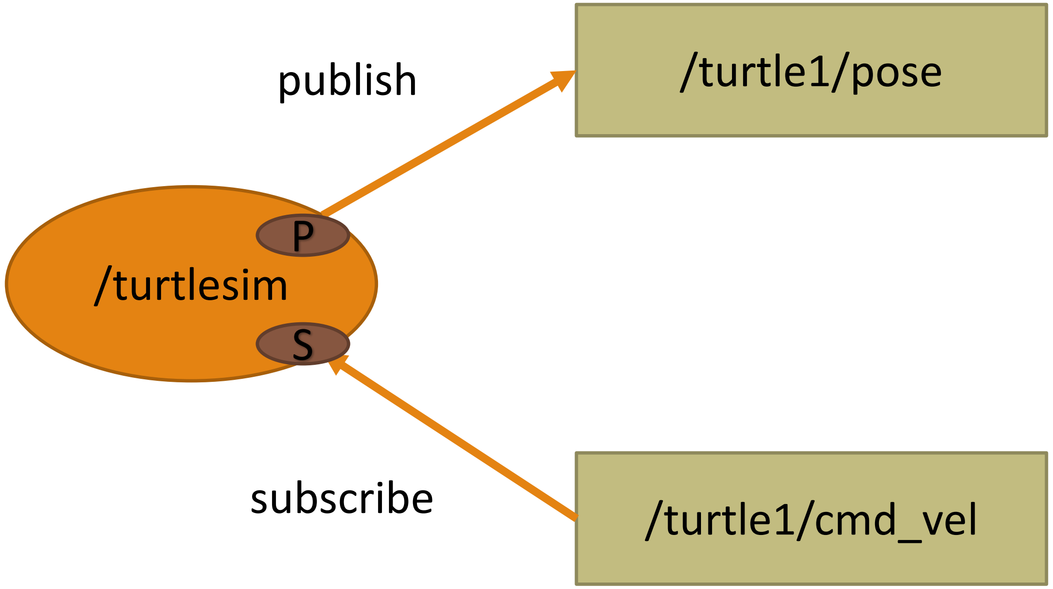
TOPIC → Communication Channel



Topics don't have to only be point-to-point communication; it can be one-to-many, many-to-one, or many-to-many.

Q2.) Should a node have both a publisher and a subscription to the same topic ?

The concept of Nodes and Topics

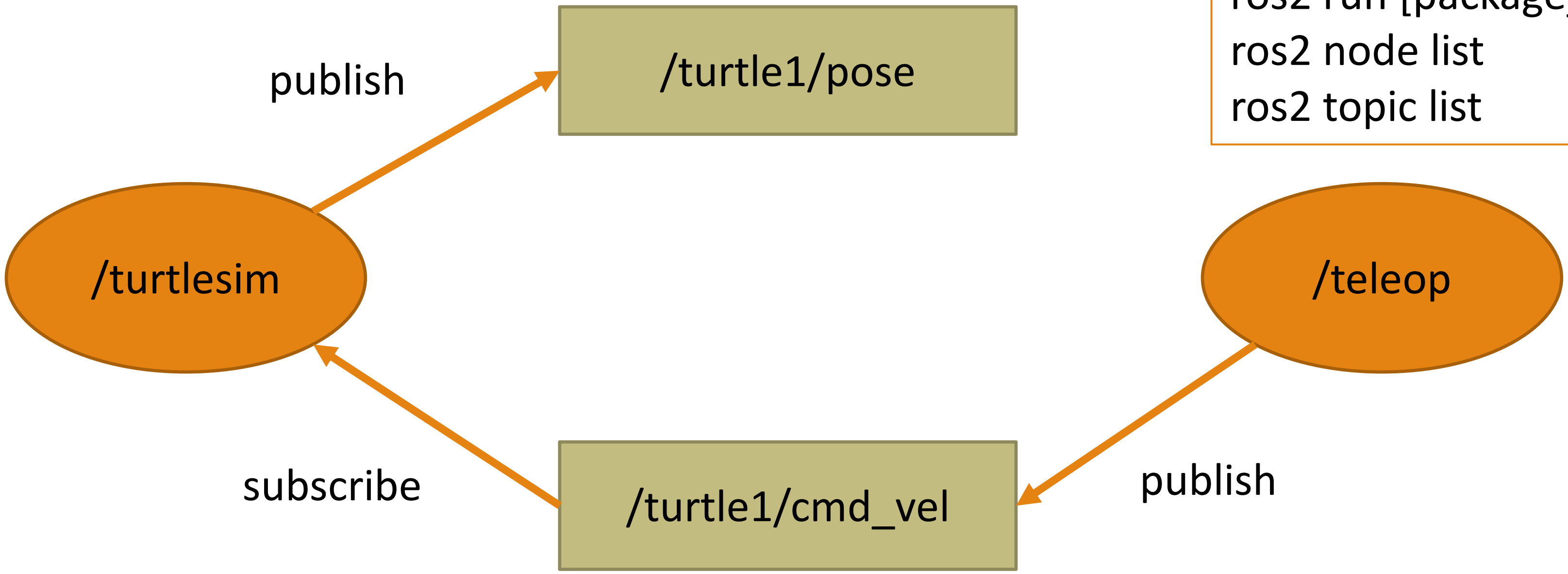


```
ros2 run [package_name] [executable]
ros2 node list
ros2 node info [node_name] --verbose
ros2 topic list
```

```
ros2 run turtlesim turtlesim_node
```

The concept of Nodes and Topics

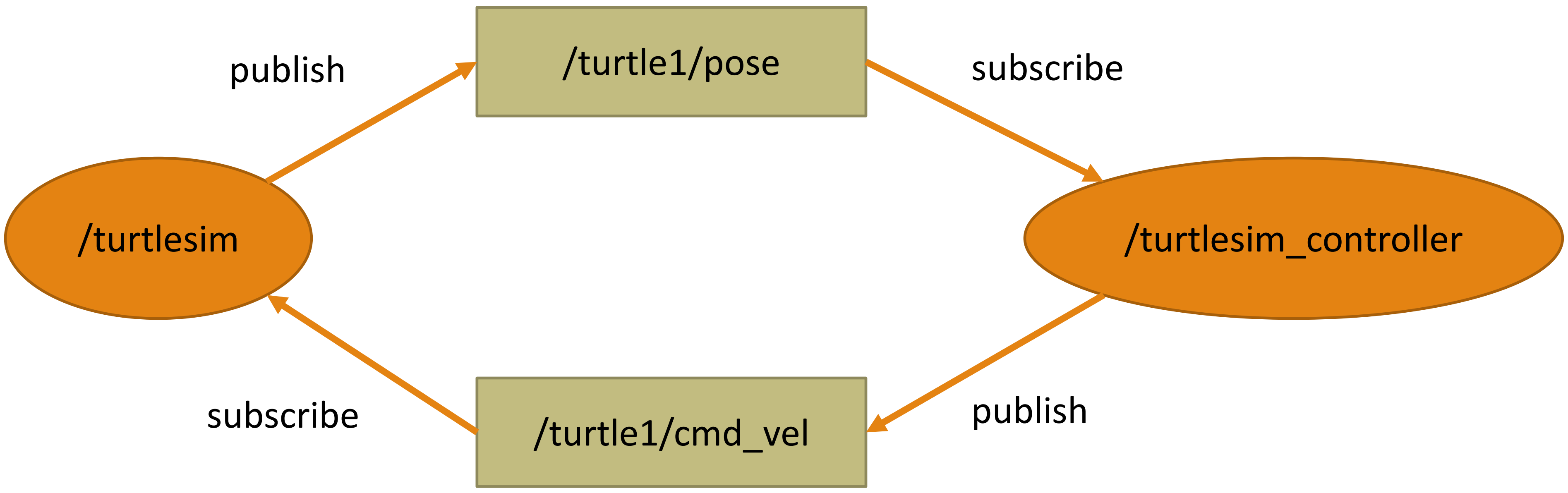
```
ros2 run [package_name] [executable]
ros2 node list
ros2 topic list
```



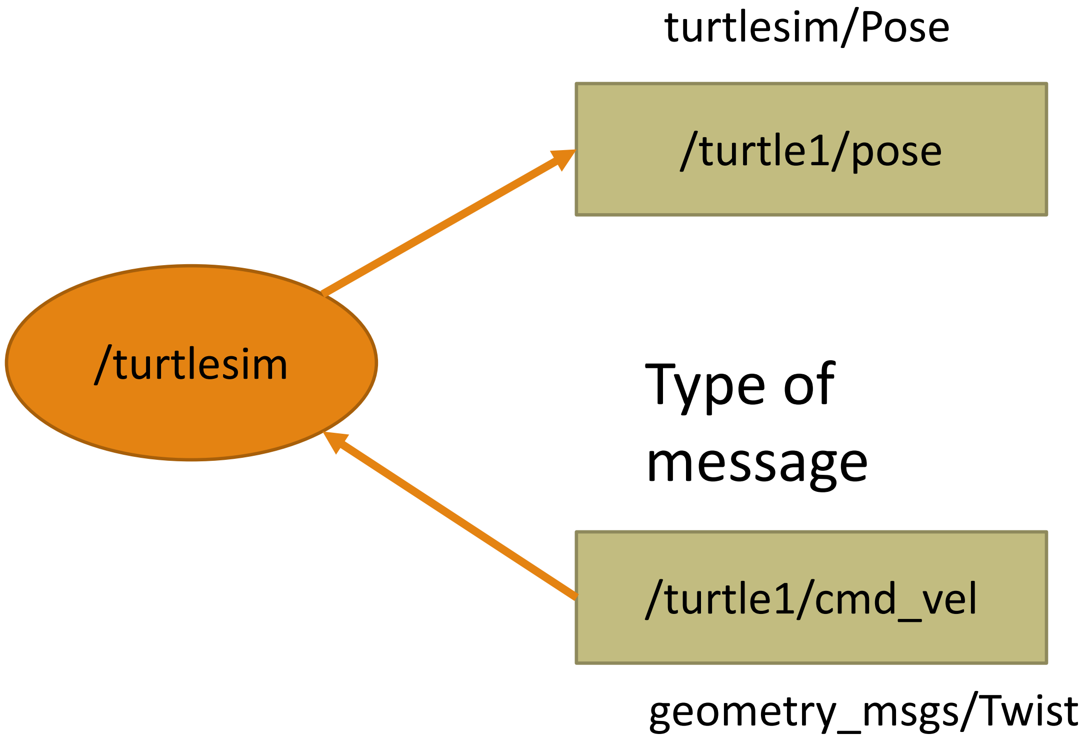
```
ros2 run turtlesim turtlesim_node
```

```
ros2 run turtlesim turtle_teleop_key
```

The concept of Nodes and Topics



The concept of message



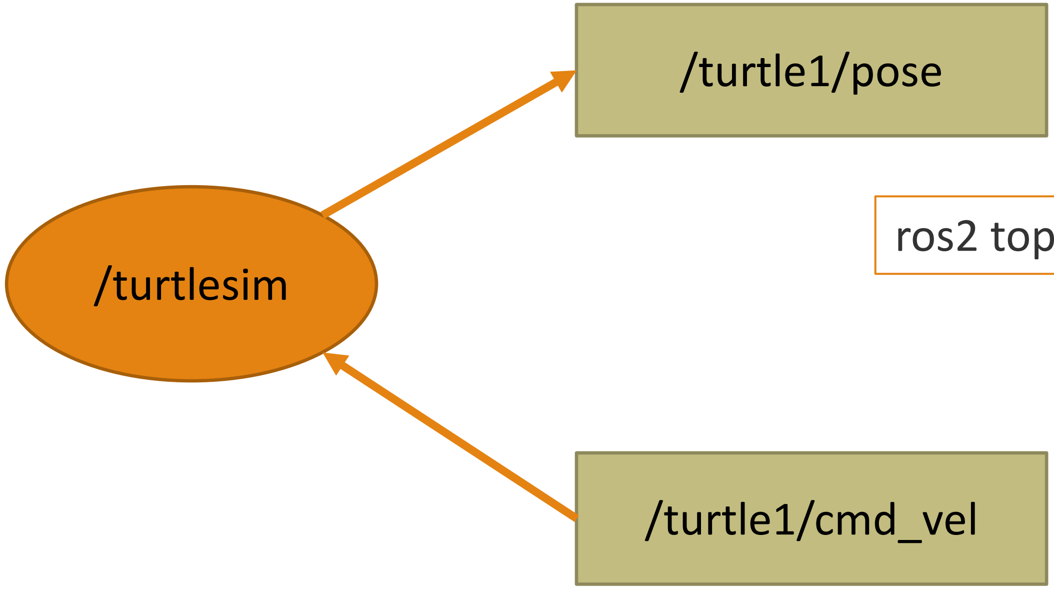
ros2 run turtlesim turtlesim_node

```
ros2 topic echo [topic_name]
ros2 topic info [topic_name]
ros2 interface show [message_type]
```

Twist :

- linear:
  - x
  - y
  - z
- angular:
  - x
  - y
  - z

Publising messages from the terminal

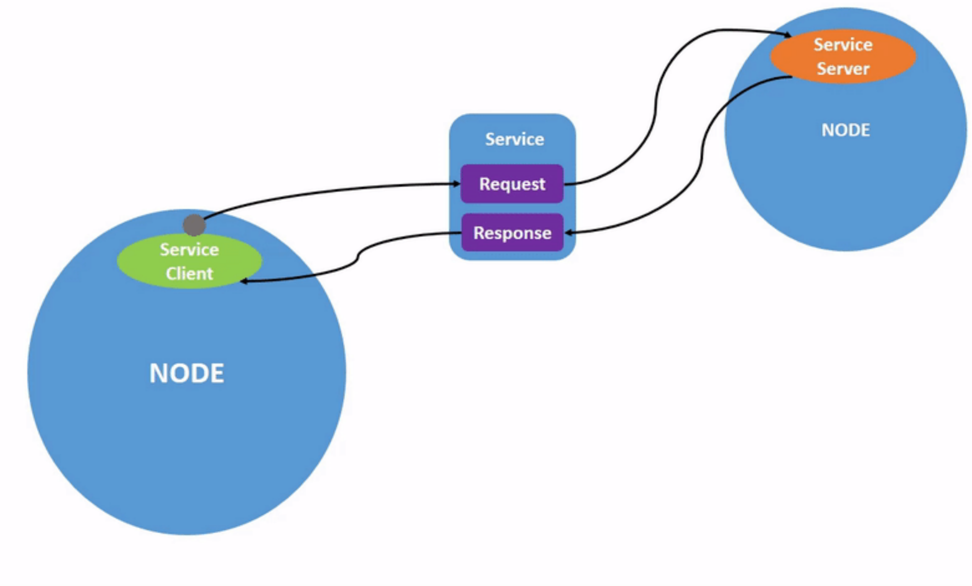


ros2 topic pub –r [rate in Hz] [topic_name] [message_type] “[data]"

YAML format

ros2 topic pub –r 10 /turtle1/cmd_vel geometry_msgs/msg/Twist {linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}

The concept of service

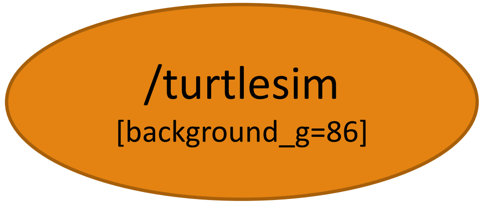


```
ros2 service list
ros2 service info [service_name]
ros2 interface show [service_type]
ros2 service call [service_name] [service_type] [request_data]
```

- 1-1 communication
- Suitable for more private communication that happens once in a while

Node can have a service server that provides service to a service client from other nodes.

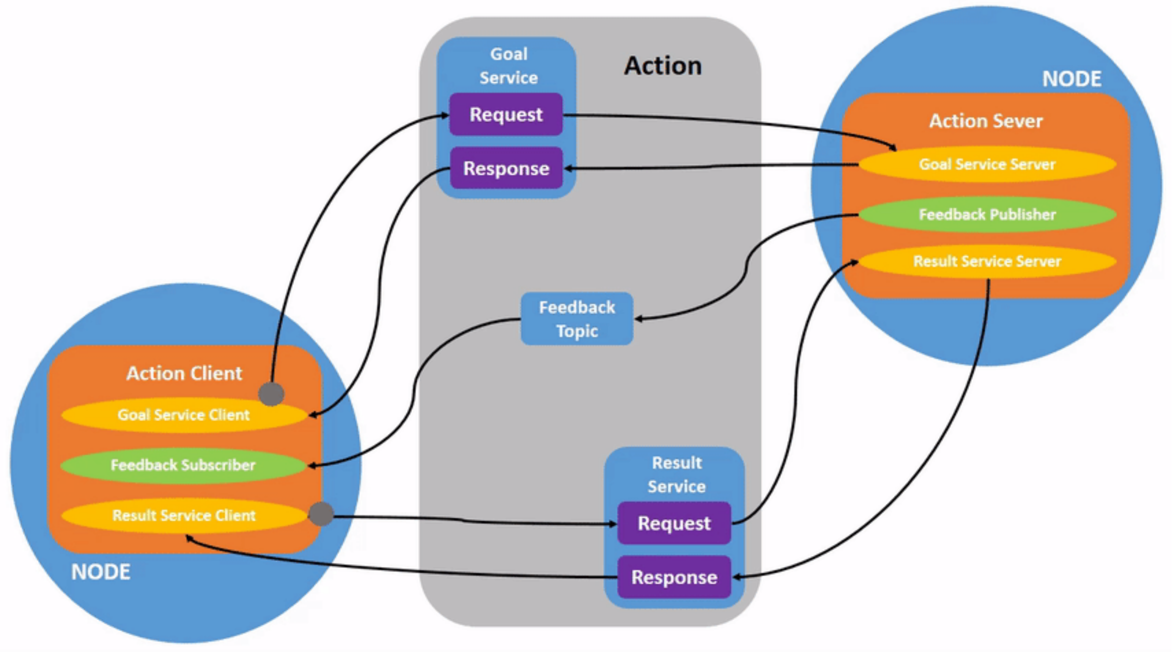
The concept of ROS2 Parameter



```
ros2 param list
ros2 param get [node_name] [parameter_name]
ros2 param set [node_name] [parameter_name] [value]
ros2 param dump [node_name]
ros2 param load [node_name] [parameter_file]
ros2 run [package_name] [executable] --ros-args –params-file [file_name]
```

Parameter is a constant that is associated with that particular node. It can be modified and accessed by terminal interface.

The concept of action



```
ros2 action list
ros2 action info [action_name]
ros2 interface show [action_type]
ros2 action send_goal [action_name] [action_type] [value] -f
```

- 1-1 communication
- Suitable for task that requires time to complete such as planning, optimizing, navigating, etc.

A node can have an action server that execute an "action", then return the result to the client.

Using existing system with ROS2

Given existing ROS2 packages, we can now know how to execute some of their functionalities via terminal command.

Let's create our own ROS2 node and packages.

Before create a ROS2 node

What will happen :

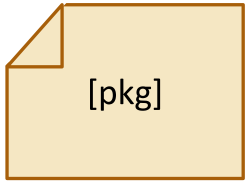
- Node will be defined in a file.
- Node will be executed via an executable.
- Node may require other data files.

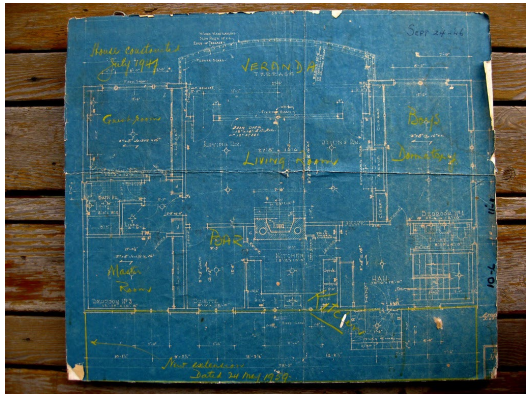
Questions :

- Where do we need to keep the definition of node ?
- Where do we need to keep the executable ?
- If the node is to be used in other computers, how can we organize the associated files systematically ?

ROS2 Package Customization

The concept of Package

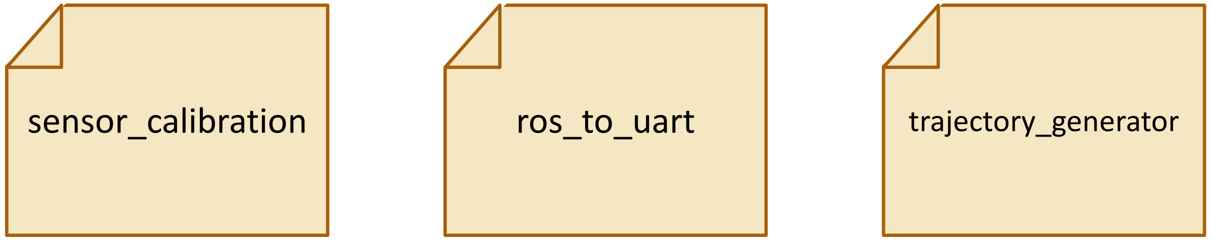




Package is a collection of organized files in a form of directory, which will used to "synthesize" nodes, launch system, etc. (usually for a specific task)

- An anology of a package would be a folder where we can keep all blue prints of a house but not the house itself.

The concept of Package



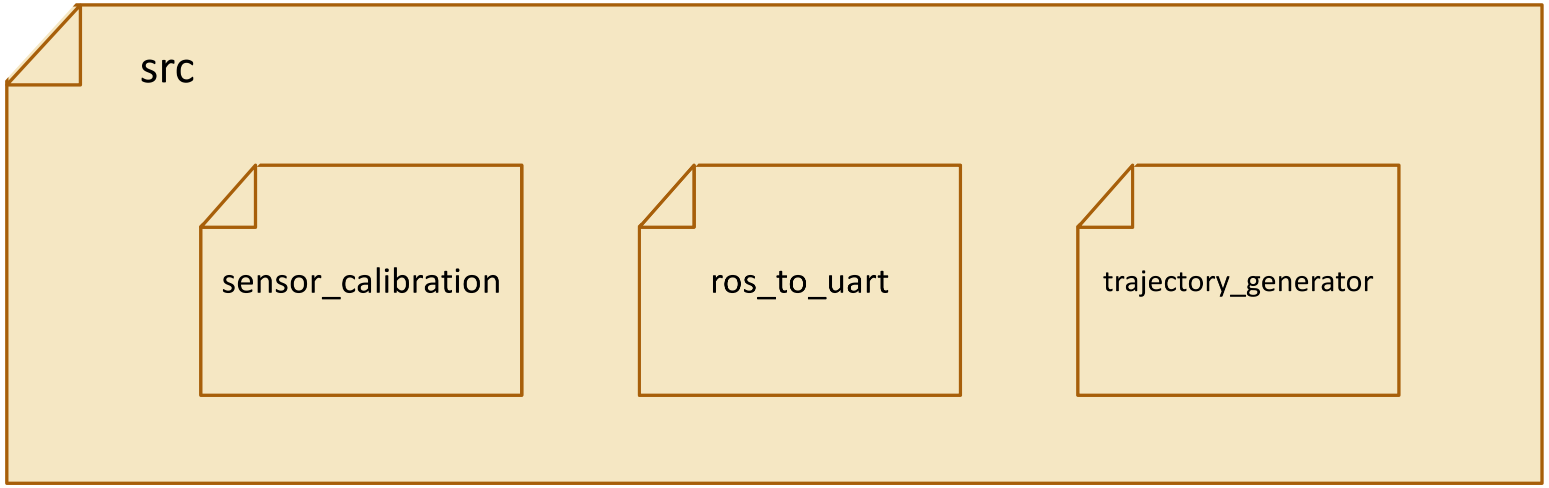
sensor_calibration

ros_to_uart

trajectory_generator

But, where do we keep our packages ?

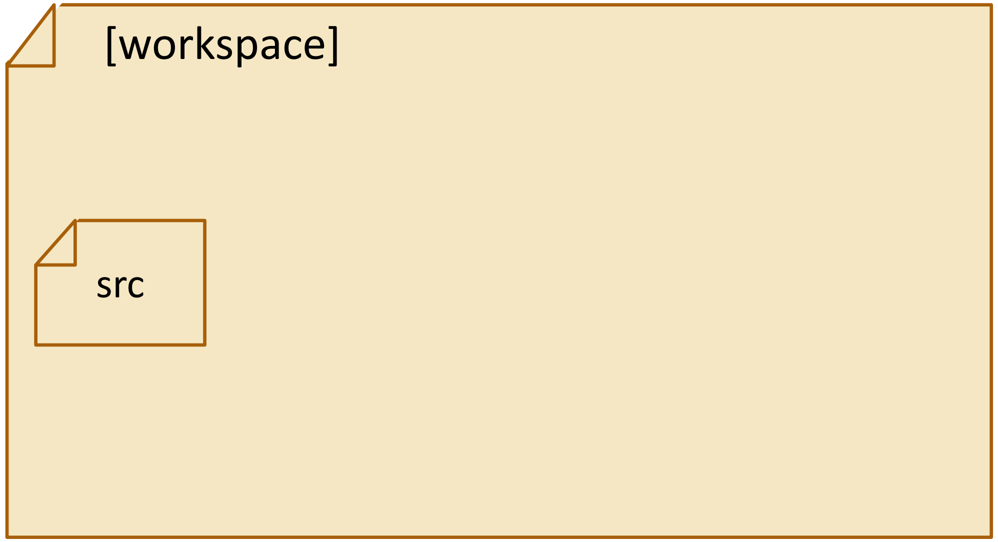
Source folder (src)



"src" directory is where we put all custom packages together.

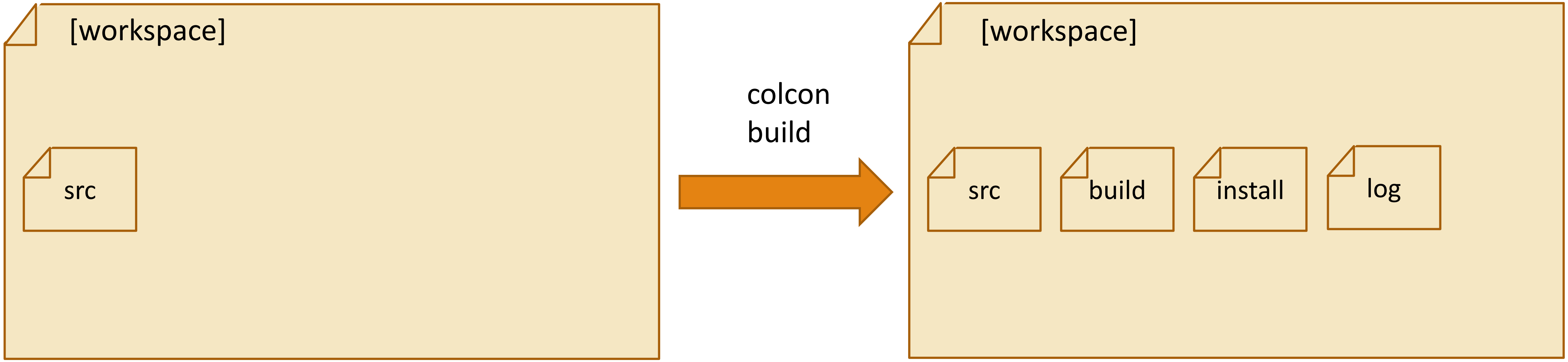
A package can be put inside another folder, which can be referred as a meta-package.

The concept of Workspace



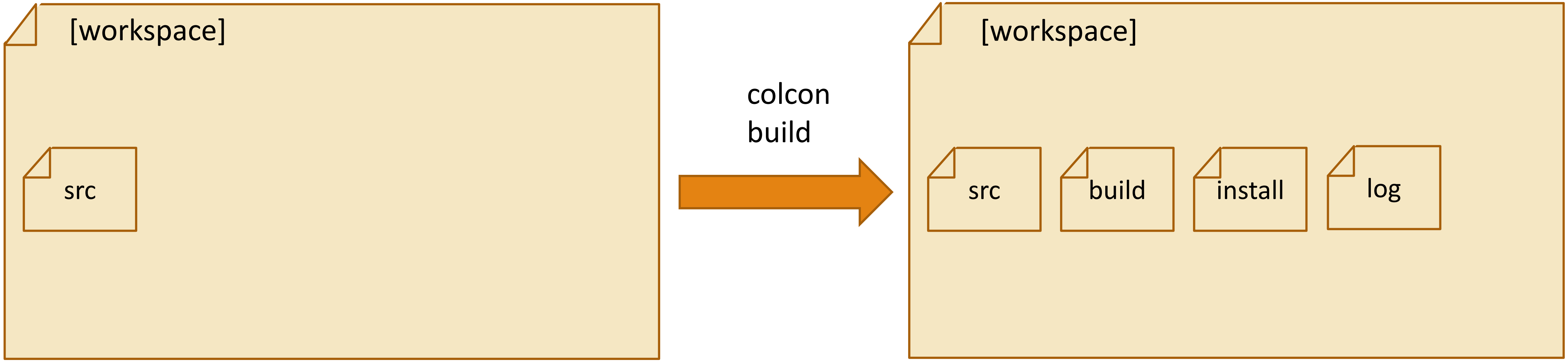
[ROS2] workspace is where we put every custom packages for a project.

Build System



"colcon build" command will build every packages in the source directory and generate 3 additional directories. When modify "src", always re-build these 3 directories. (with an exception of using symlink install)

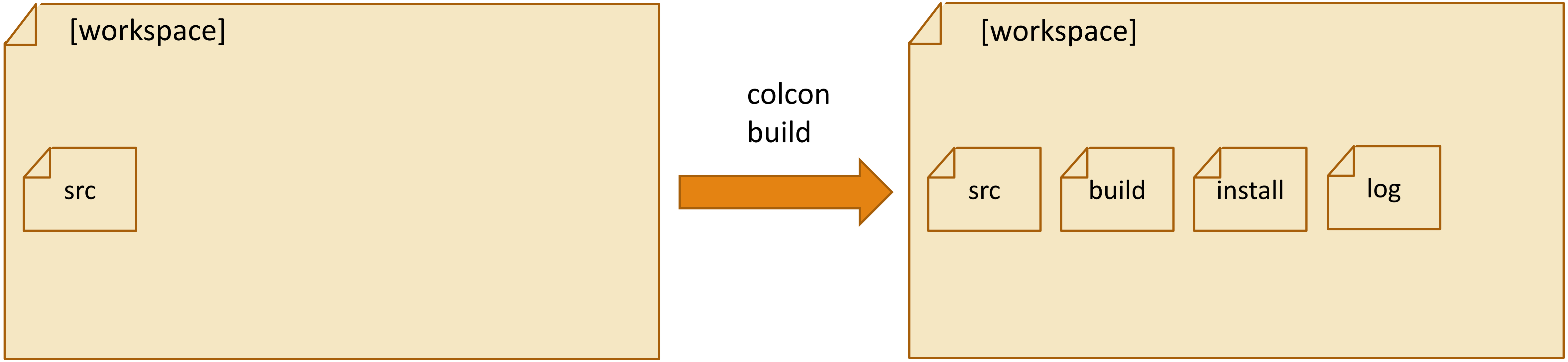
Build System



"Install" directory is the location of codes that will be used by the ROS2 system. Therefore, only modifying "src" will not change the behavior of your system in run time.

Build System

You must not create workspace inside another workspace



"Install" directory is the location of codes that will be used by the ROS2 system. Therefore, only modifying "src" will not change the behavior of your system in run time.

Creating a new workspace

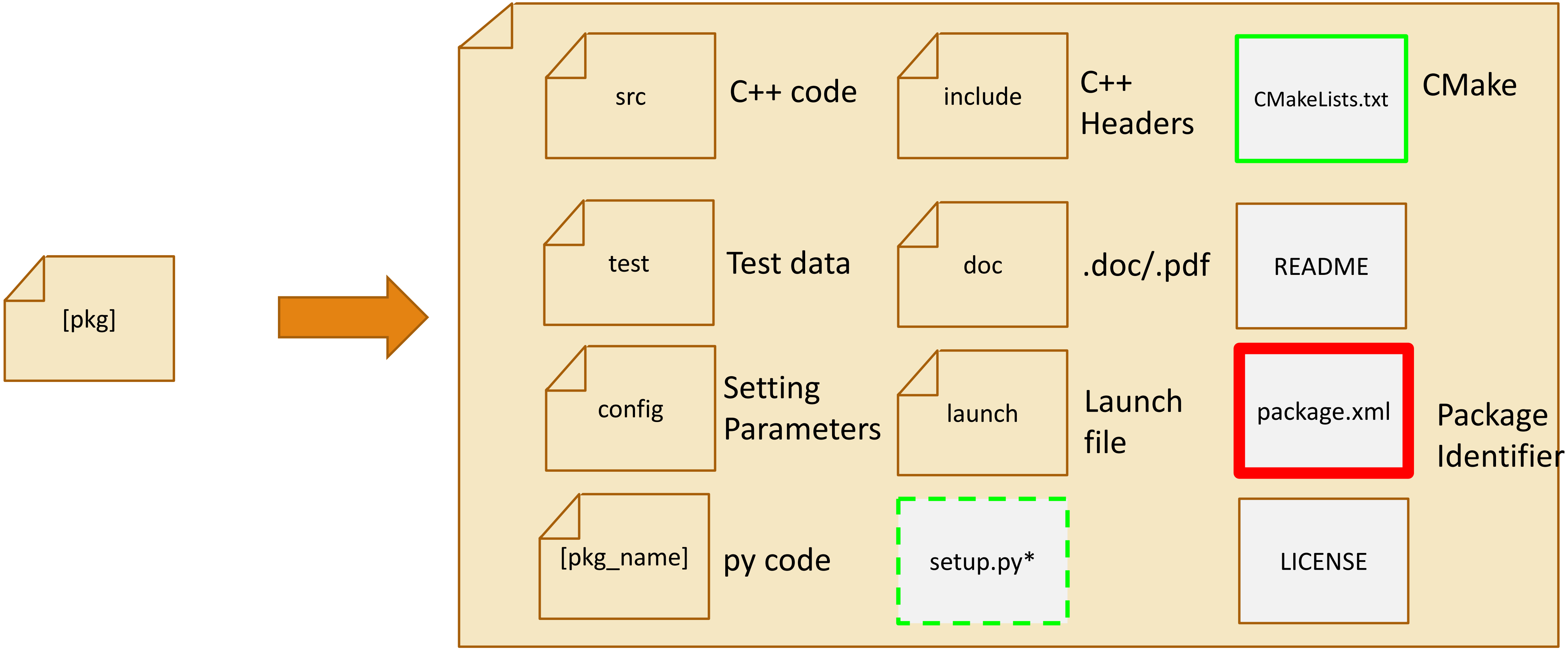
Create and build a new workspace

```
>> mkdir –p ~/[xxx]_ws/src
>> cd ~/[xxx]_ws
>> colcon build
```

Adding workspace to .bashrc

```
source ~/[xxx]_ws/install/local_setup.bash
```

Package Layout



https://automaticaddison.com/organizing-files-and-folders-inside-a-ros-2-package/

Creating a ROS Package

```
>> cd ~/[xxx]_ws/src
>> ros2 pkg create --build-type ament_python [package_name]
```

Package with Python

```
>> cd ~/[xxx]_ws/src
>> ros2 pkg create --build-type ament_cmake [package_name]
```

Package with C++

The package that we create must be in the src directory of our workspace.

WCreating a ROS Packagehat if we have both C++ and Python files ?

Creating my_first_pkg

```
>> cd ~/[xxx]_ws/src
>> ros2 pkg create --build-type ament_python my_first_pkg
>> cd ..
>> colcon build --packages-select my_first_pkg

>> cd src/my_first_pkg
>> code package.xml

>> cd my_first_pkg
>> code my_first_script.py
```

https://ocs.ros.org/en/diamondback/api/licenses.html

Creating my_first_pkg

```
>> cd ~/[xxx]_ws/src
>> ros2 pkg create --build-type ament_python my_first_pkg
>> cd my_first_pkg
>> code my_first_script.py
```

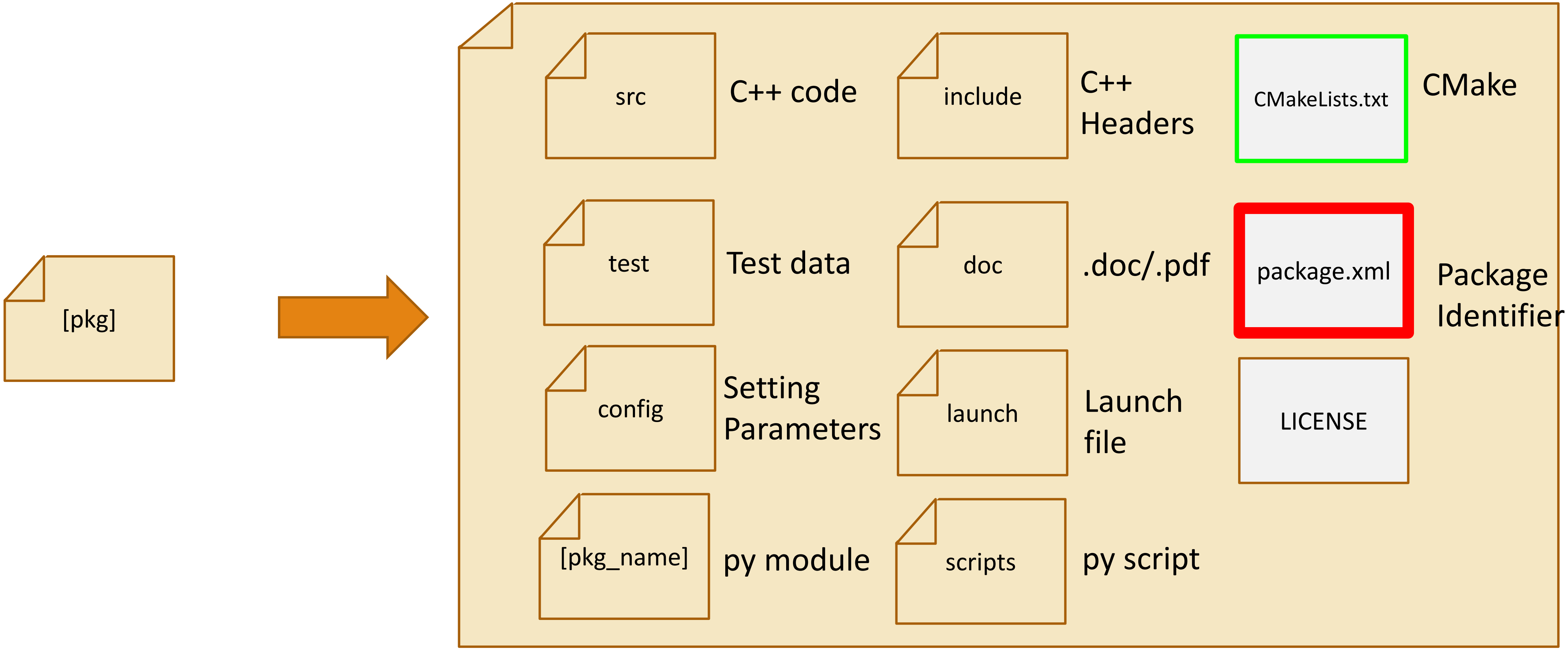
What if we have both C++ and Python files ?

https://ocs.ros.org/en/diamondback/api/licenses.html

Using ROS_pkg_cpp_py

```
>> cd ~
>> git clone https://github.com/tchoopojcharoen/ROS2_pkg_cpp_py
>> source ROS2_pkg_cpp_py/install_pkg.bash {YOUR_WORKSPACE} {PACKAGE_NAME}
```

Package Layout



https://automaticaddison.com/organizing-files-and-folders-inside-a-ros-2-package/

Structure of CMakeLists.txt for building a package

```
# C++ Dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)

# Python Dependencies
find_package(ament_cmake_python REQUIRED)
find_package(rclpy REQUIRED)

# add C++ include
include_directories(include)

# add PythonModule
ament_python_install_package($(PROJECT_NAME))
```

```
add_executable(my_exe_name src/my_exe.cpp)
ament_target_dependencies(my_exe_name rclcpp)

install(TARGETS
           my_exe_name
         DESTINATION lib/$(PROJECT_NAME)
)

install(PROGRAMS
         scripts/my_scripts.py
         scripts/my_scripts_2.py
         DESTINATION share/$(PROJECT_NAME)
)

install(DIRECTORY
         launch
         config
         DESTINATION share/$(PROJECT_NAME)
)
ament_package()
```

compile and create executable

manage dependencies

put executable in install

put scripts in install

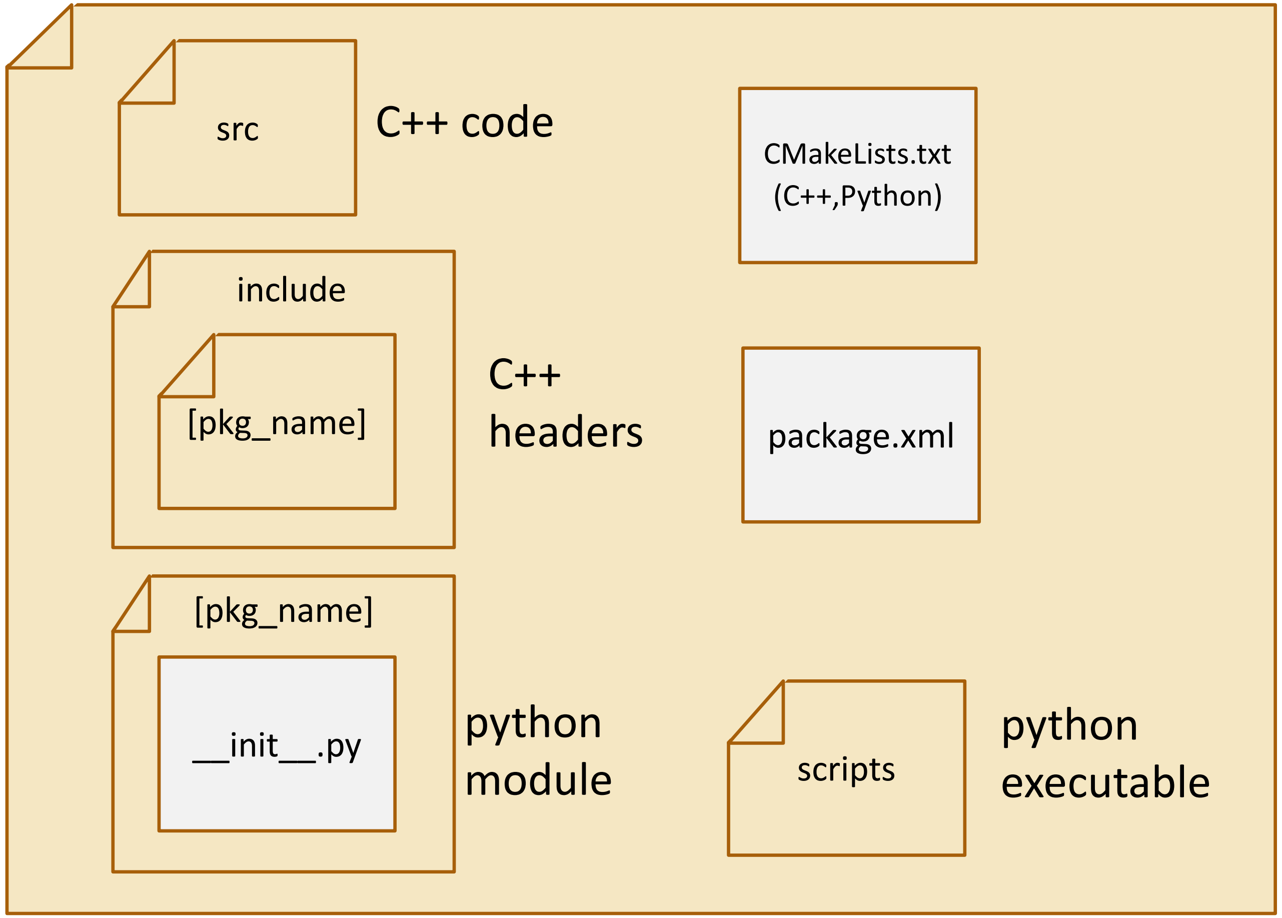
put directories in install

Creating Package Manually (C++ & Python)

1. Create a package with ament_cmake
2. Create a new folder with the same name as the package and add an empty python file with the name __init__.py
3. Make sure that all scripts consists of (#!/usr/bin/python3) in the beginning of the file
4. Apply "chmod +x" to all Python executables

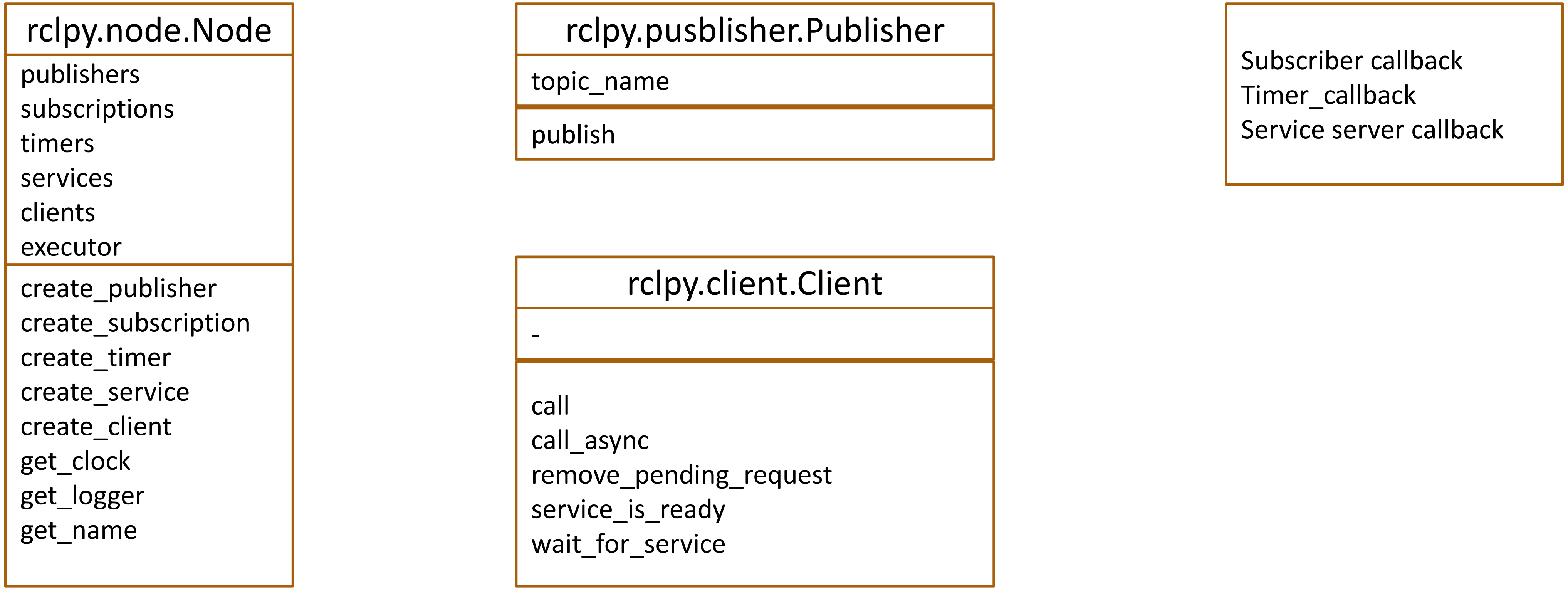
```
>> cd [path to workspace
>> find src/[package_name]/scripts -exec chmod +x {} \;
```

5. Add additional folders such as launch, config, etc
6. Modify CMakeLists.txt accordingly
   - Python executable
   - Folder
   - Custom Interface



ROS2 Node Programming with RCLPY

Node class in ROS Client Library for Python (RCLPY)



rclpy.node.Node

publishers
subscriptions
timers
services
clients
executor

create_publisher
create_subscription
create_timer
create_service
create_client
get_clock
get_logger
get_name

rclpy.pusblisher.Publisher

topic_name

publish

rclpy.client.Client

-

call
call_async
remove_pending_request
service_is_ready
wait_for_service

Subscriber callback
Timer_callback
Service server callback

General procedure on "running" a node using RCLPY

1. Initialize rclpy using 'rclpy.init'
2. Instantiate a "Node" object
3. Spin node using rclpy.spin or spin once in a while-loop
4. If the processs exits the loop, de-construct the node
5. Shut down rclpy using rclpy.shutdown

Inheriting a Node class from RCLPY

```
from rclpy.Node import Node

class NewNode(Node):
    def __init__(self):
        super().__init__('node_name')
def main(args=None):
    rclpy.init(args=args)
    node = NewNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()
```

Import the superclass "Node"

"NewNode" inherits from "Node"

Running a node of class "NewNode"

Timer, Publisher, & Subscription

Attaching a timer to the node

```
class NewNode(Node):
    def __init__(self):
        super().__init__('node_name')
        self.timer = self.create_timer(1.0,timer_callback)
def timer_callback():
    print('Hello World')
```

Period of the timer

Attaching the callback

Defining callback as a function

Attaching a timer to the node

```
class NewNode(Node):
    def __init__(self):
        super().__init__('node_name')
        self.count = 0
        self.timer = self.create_timer(1.0,self.timer_callback)
    def timer_callback(self):
        print(self.count)
        self.count = self.count + 1
```

Defining new attribute

Make a value persist by accessing the attribuate of te instance

Defining callback as a method of the class

Attaching a publisher and publishing a message in a timer

```
from std_msgs/msg import Int32

class NewNode(Node):
    def __init__(self):
        super().__init__('node_name')
        self.count = 0
        self.timer = self.create_timer(1.0,self.timer_callback)
        self.pub = self.create_publisher(Int32,'output',10)
    def timer_callback(self):
        self.count = self.count + 1
        msg = Int32()
        msg.data = self.count
        self.pub.publish(msg)
```

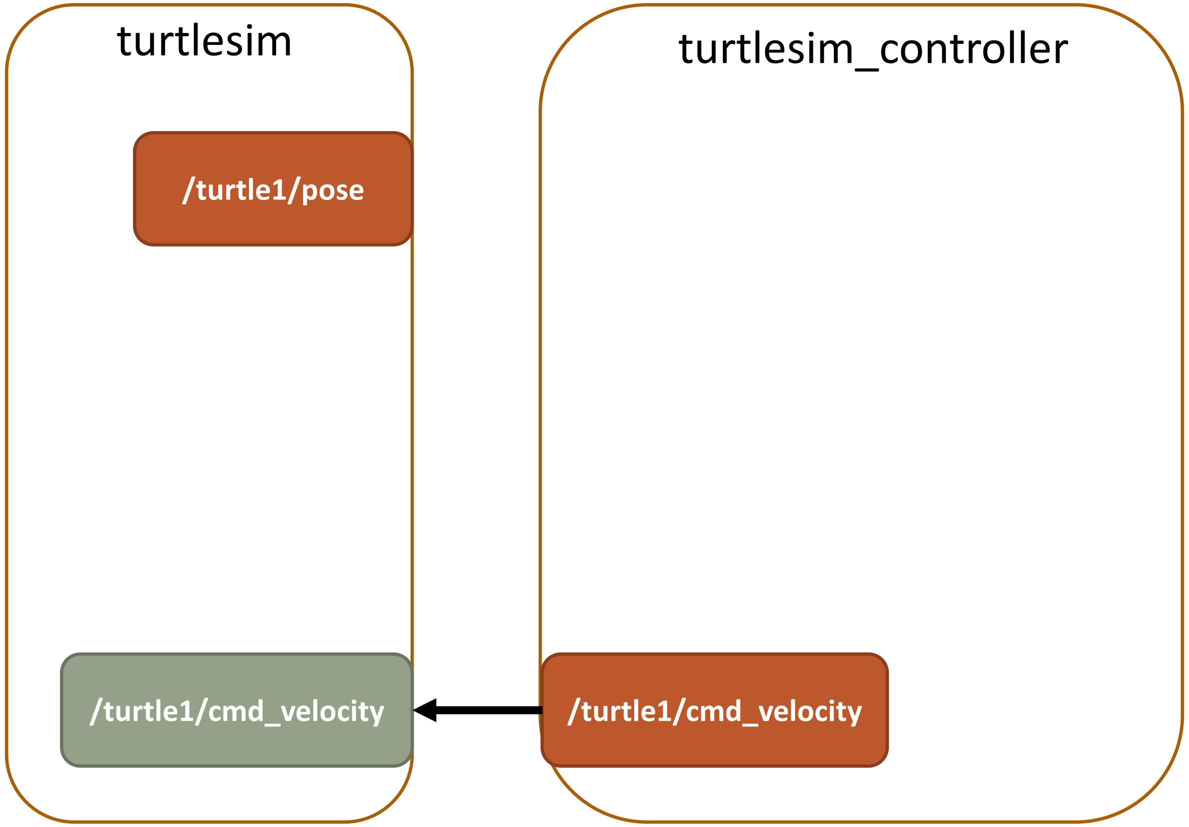
Import type of message

create a publisher

Instantiate a new message

publish a message

Exercise 0 : Driving in Circle



Create a node that allows the associated turtle to drive in circle.

Attaching a subscription and its callback

```
from std_msgs/msg import Int32

class NewNode(Node):
    def __init__(self):
        super().__init__('node_name')
        self.pub = self.create_publisher(Int32,'output',10)
        self.sub = self.create_subscription(Int32,'input',self.sub_callback,10)
    def subscription_callback(self,msg):
        pub_msg = Int32()
        pub_msg.data = msg.data + 1
        Self.pub.publish(pub_msg)
```

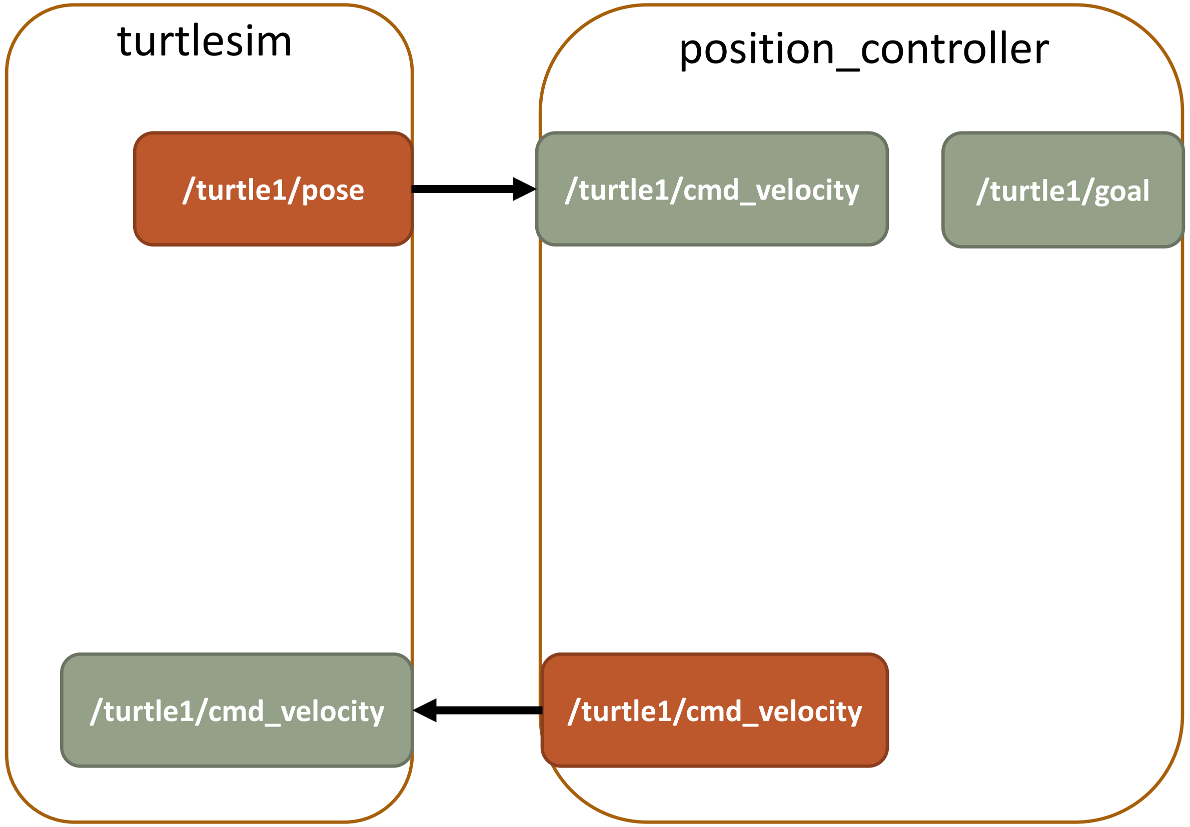
Import type of message

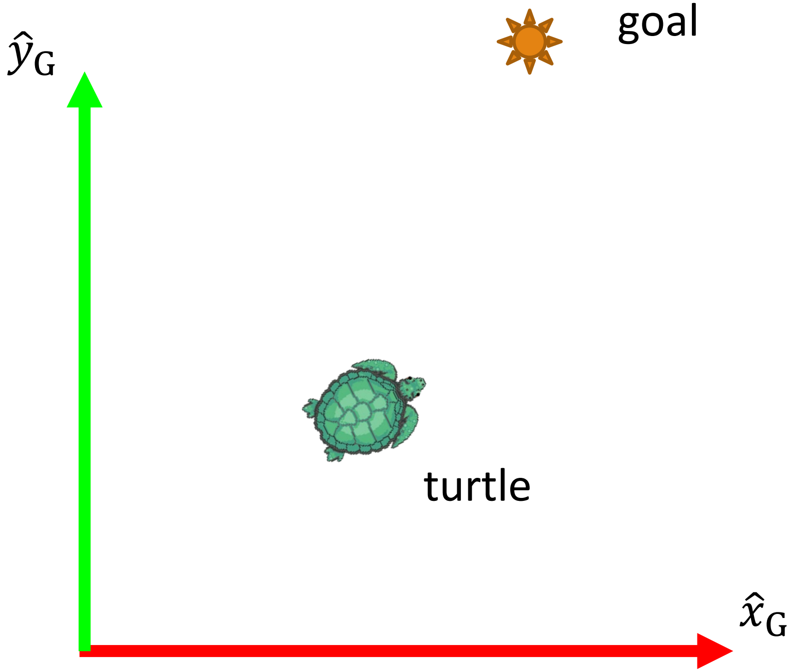
create a subscription

Argument format of subscription callback

These lines will be executed when 'input' is published by other node(s)

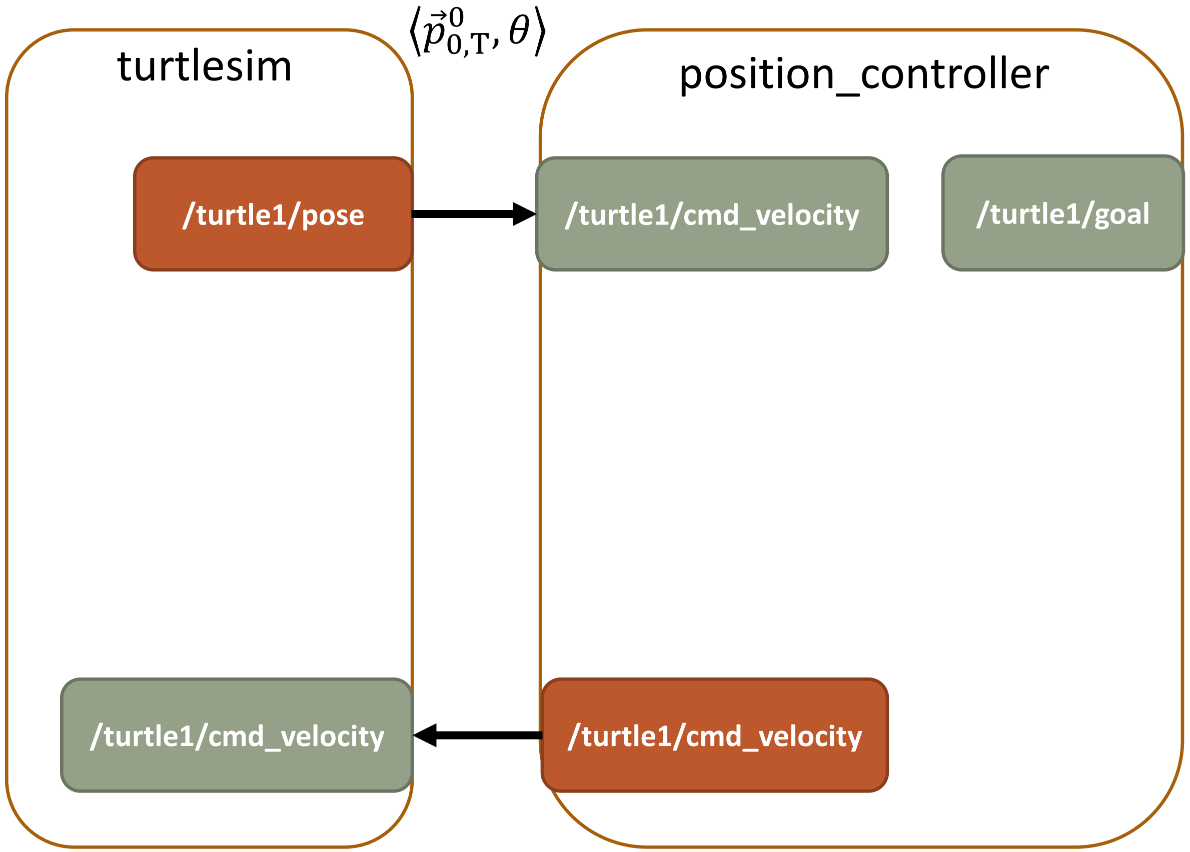
Exercise 1: Position Control

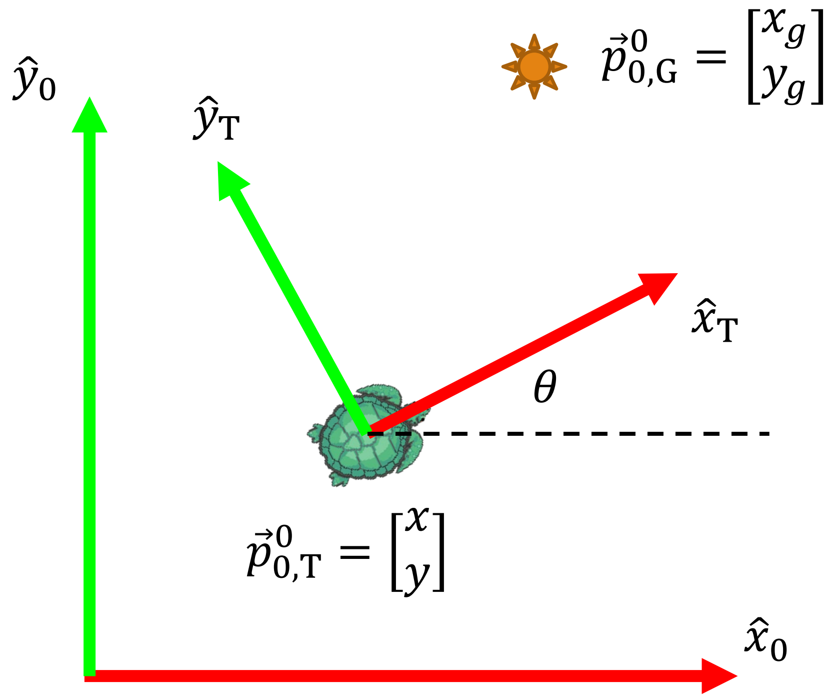




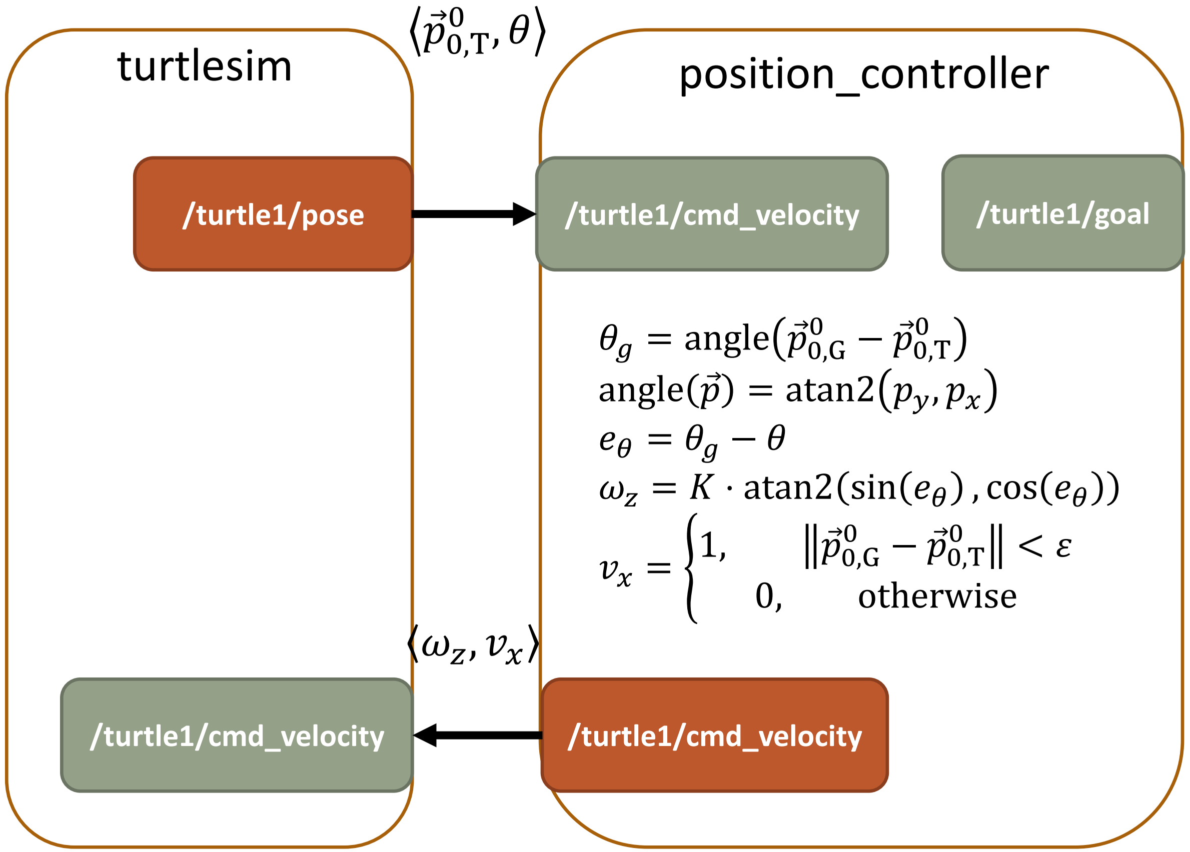
Objective : The turtle must reach the goal.

Exercise 1: Position Control




$\hat{y}_0$
$\hat{y}_\mathrm{T}$
$\vec{p}^{\,0}_{0,\mathrm{G}} = \begin{bmatrix} x_g \\ y_g \end{bmatrix}$
$\hat{x}_\mathrm{T}$
θ
$\vec{p}^{\,0}_{0,\mathrm{T}} = \begin{bmatrix} x \\ y \end{bmatrix}$
$\hat{x}_0$

Exercise 1: Position Control



$\langle \vec{p}^{\,0}_{0,\mathrm{T}}, \theta \rangle$

turtlesim

/turtle1/pose

/turtle1/cmd_velocity

position_controller

/turtle1/cmd_velocity

/turtle1/goal

$$\theta_g = \mathrm{angle}\left(\vec{p}^{\,0}_{0,\mathrm{G}} - \vec{p}^{\,0}_{0,\mathrm{T}}\right)$$

$$\mathrm{angle}(\vec{p}) = \mathrm{atan2}\left(p_y, p_x\right)$$

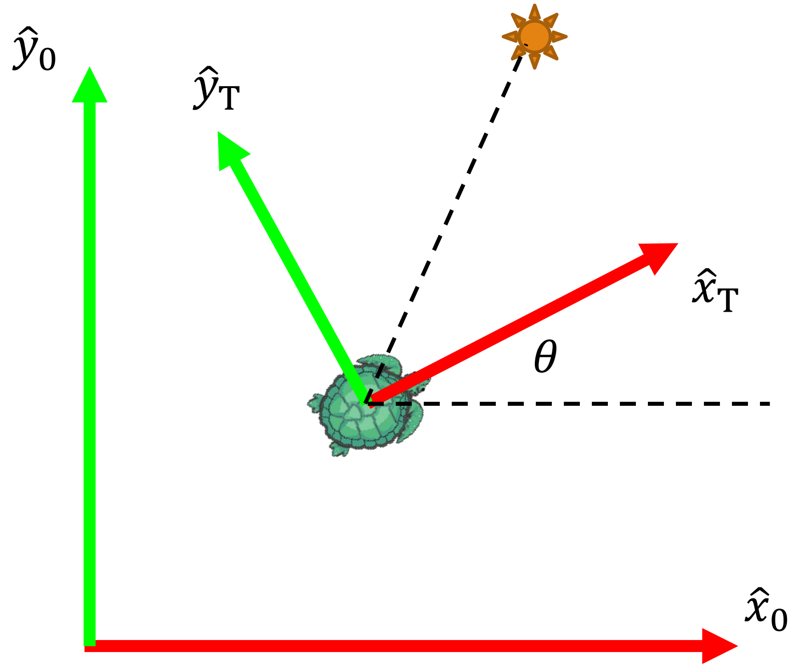
$$e_\theta = \theta_g - \theta$$

$$\omega_z = K \cdot \mathrm{atan2}(\sin(e_\theta), \cos(e_\theta))$$

$$v_x = \begin{cases} 1, & \left\|\vec{p}^{\,0}_{0,\mathrm{G}} - \vec{p}^{\,0}_{0,\mathrm{T}}\right\| < \varepsilon \\ 0, & \text{otherwise} \end{cases}$$

$\langle \omega_z, v_x \rangle$

/turtle1/cmd_velocity



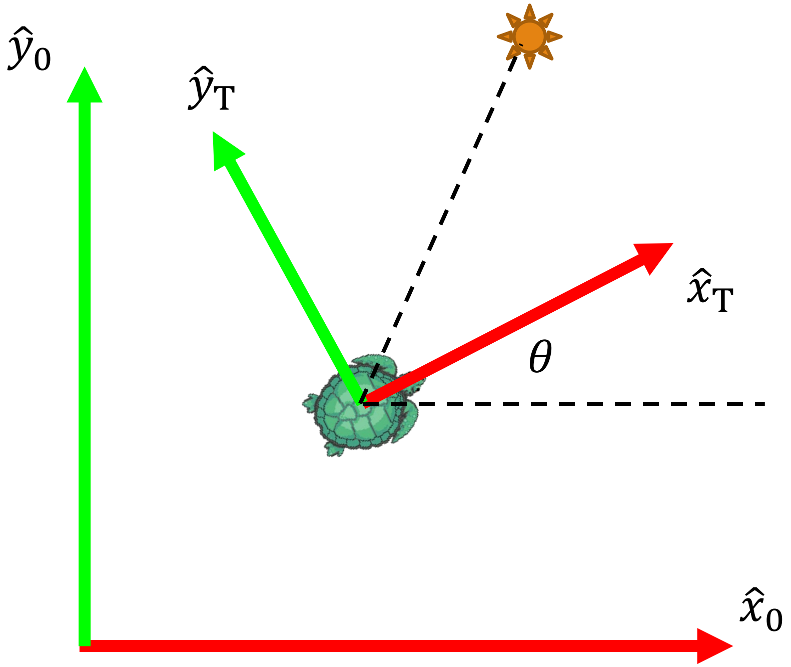
Go-to-goal Control Policy

$$\omega_z = K \cdot \operatorname{atan2}(\sin(e_\theta), \cos(e_\theta))$$

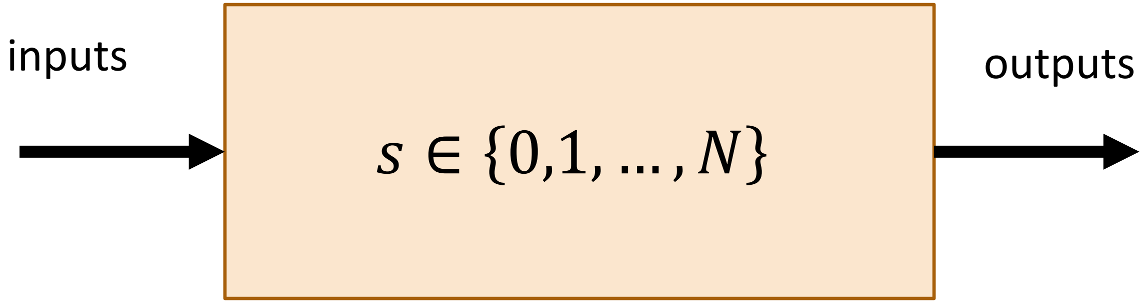
$$v_x = \begin{cases} 1, & \left\|\vec{p}^{\,0}_{0,\mathrm{G}} - \vec{p}^{\,0}_{0,\mathrm{T}}\right\| < \varepsilon \\ 0, & \text{otherwise} \end{cases}$$

where

$$e_\theta = \theta_g - \theta$$
$$\theta_g = \operatorname{angle}\left(\vec{p}^{\,0}_{0,\mathrm{G}} - \vec{p}^{\,0}_{0,\mathrm{T}}\right)$$
$$\operatorname{angle}(\vec{p}) = \operatorname{atan2}\left(p_y, p_x\right)$$



Finite State Machine



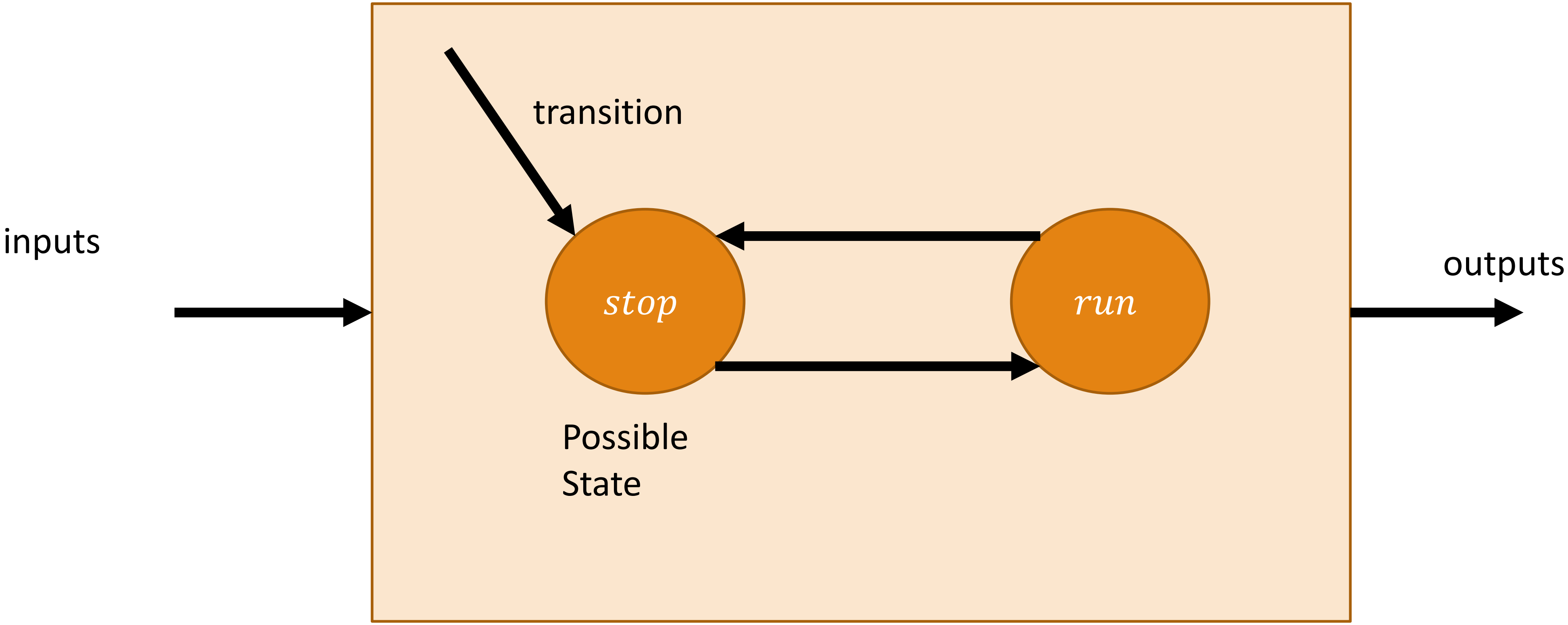
FSM consists of discrete amount of possible states .

There 2 types of updating scheme for an FSM.

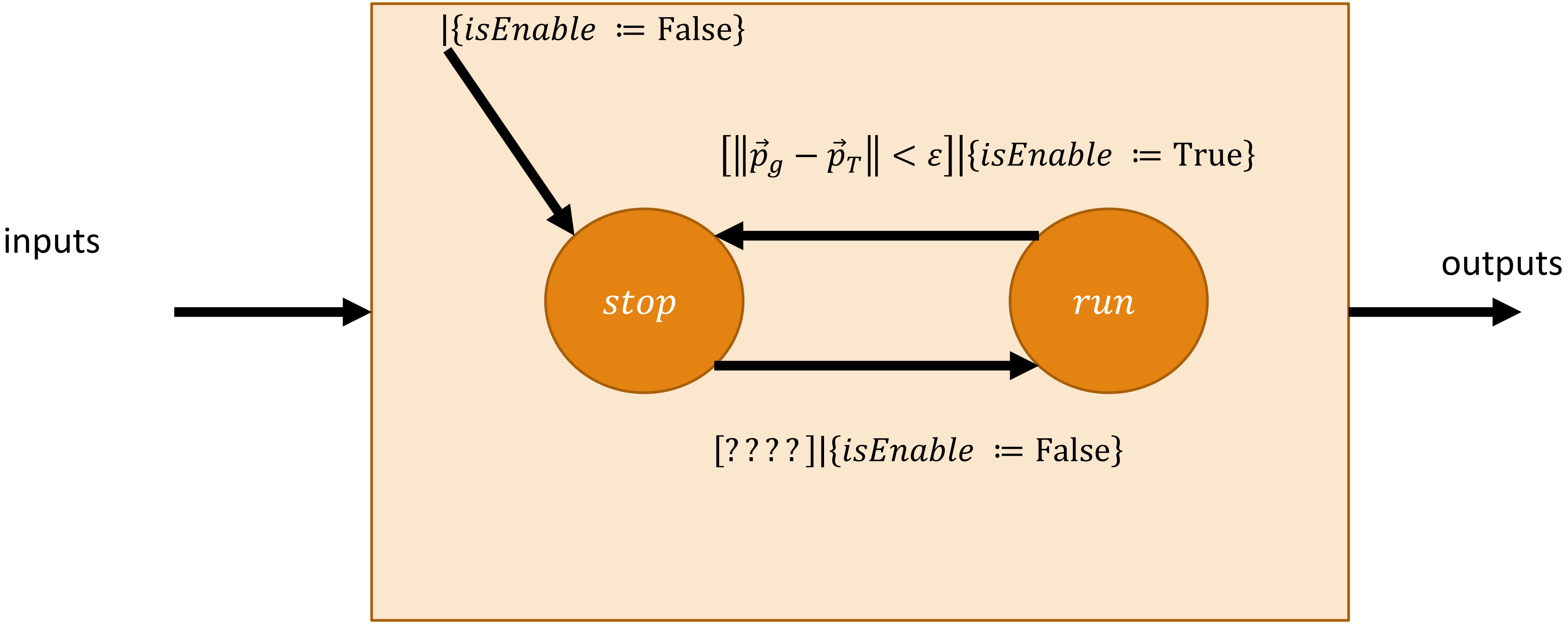
1.) Updating at fixed time interval (Time-driven)

2.) Updating when particular events occur (Event-driven)

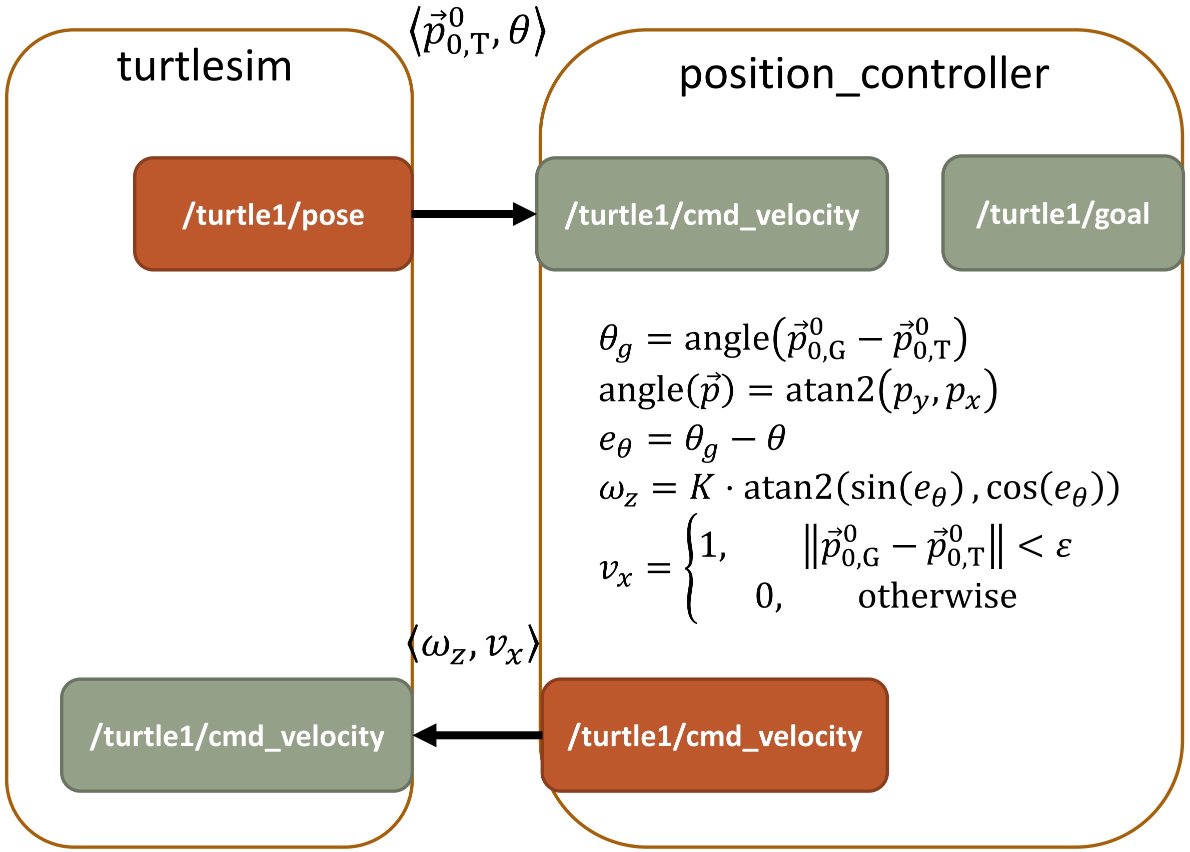
Finite State Machine



Finite State Machine


$|\{isEnable \coloneqq \text{False}\}$
$[\|\vec{p}_g - \vec{p}_T\| < \varepsilon]|\{isEnable \coloneqq \text{True}\}$
inputs
$stop$
run
outputs
$[????]|\{isEnable \coloneqq \text{False}\}$

Exercise 1: Position Control



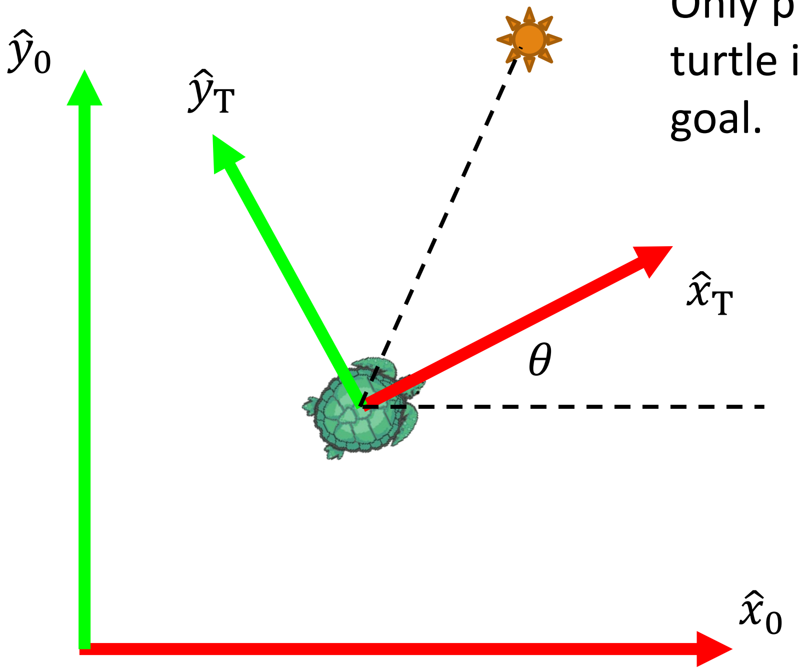
$$\theta_g = \mathrm{angle}\left(\vec{p}^{\,0}_{0,\mathrm{G}} - \vec{p}^{\,0}_{0,\mathrm{T}}\right)$$

$$\mathrm{angle}(\vec{p}) = \mathrm{atan2}\left(p_y, p_x\right)$$

$$e_\theta = \theta_g - \theta$$

$$\omega_z = K \cdot \mathrm{atan2}(\sin(e_\theta), \cos(e_\theta))$$

$$v_x = \begin{cases} 1, & \left\|\vec{p}^{\,0}_{0,\mathrm{G}} - \vec{p}^{\,0}_{0,\mathrm{T}}\right\| < \varepsilon \\ 0, & \text{otherwise} \end{cases}$$



Only publish when the turtle is far from the goal.

Creating multiple nodes that do similar tasks

What if we want to create anoter controller that follow another turtle ?

Should we refine everything ?

Object-Oriented Programming

Shared Features & Functionality

TurtleFollower

Use go-to-goal to follow a turtle

ViaPointFollower

Use go-to-goal to follow a turtle

Shared Features & Functionality

- Knowing the location of the associated turtle
- Sending control input to turtlesim
- Using the same control law

TurtleFollower
ViaPointFollower

Shared Features & Functionality

- Knowing the location of the associated turtle
- Sending control input to turtlesim
- Using the same control law

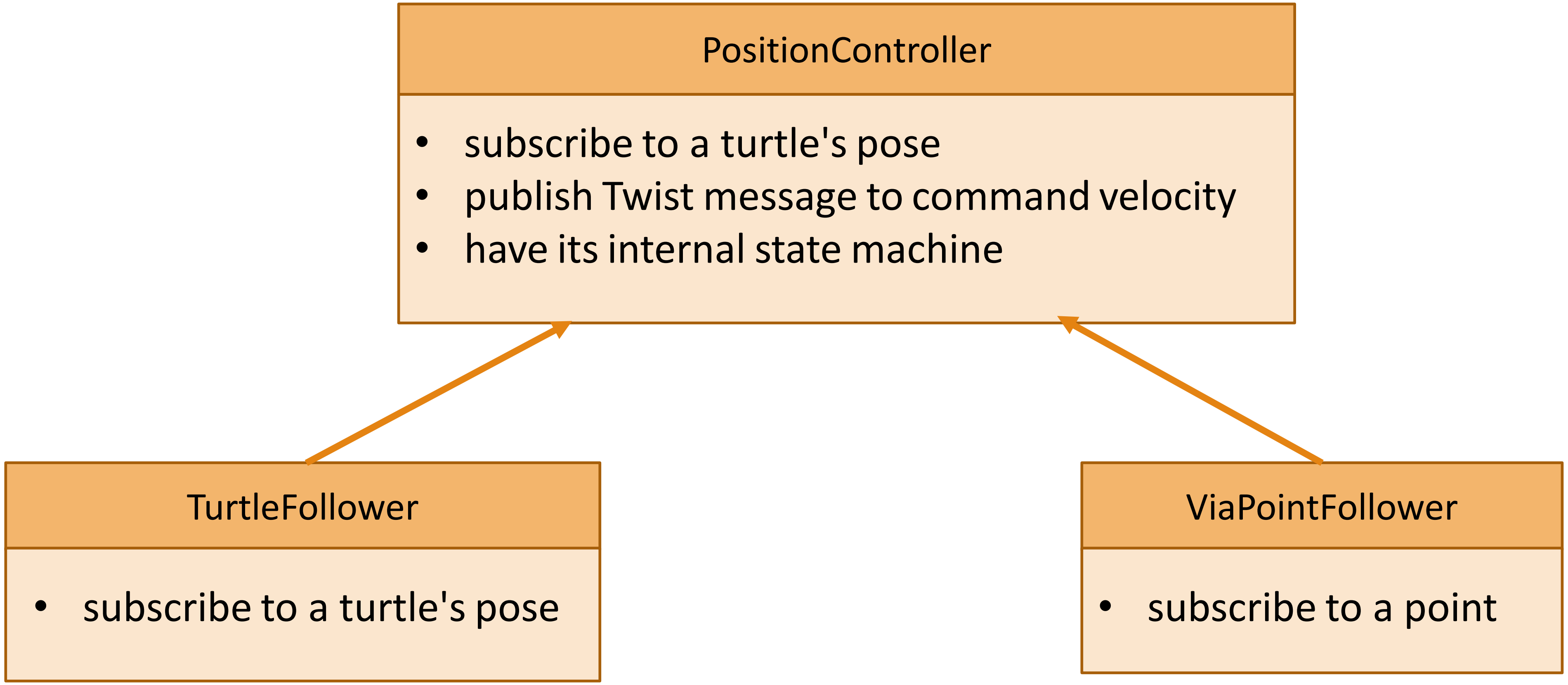
TurtleFollower

- The goal is turtle's pose

ViaPointFollower

- The goal is a position.

Object-Oriented Programming



PositionController

- subscribe to a turtle's pose
- publish Twist message to command velocity
- have its internal state machine

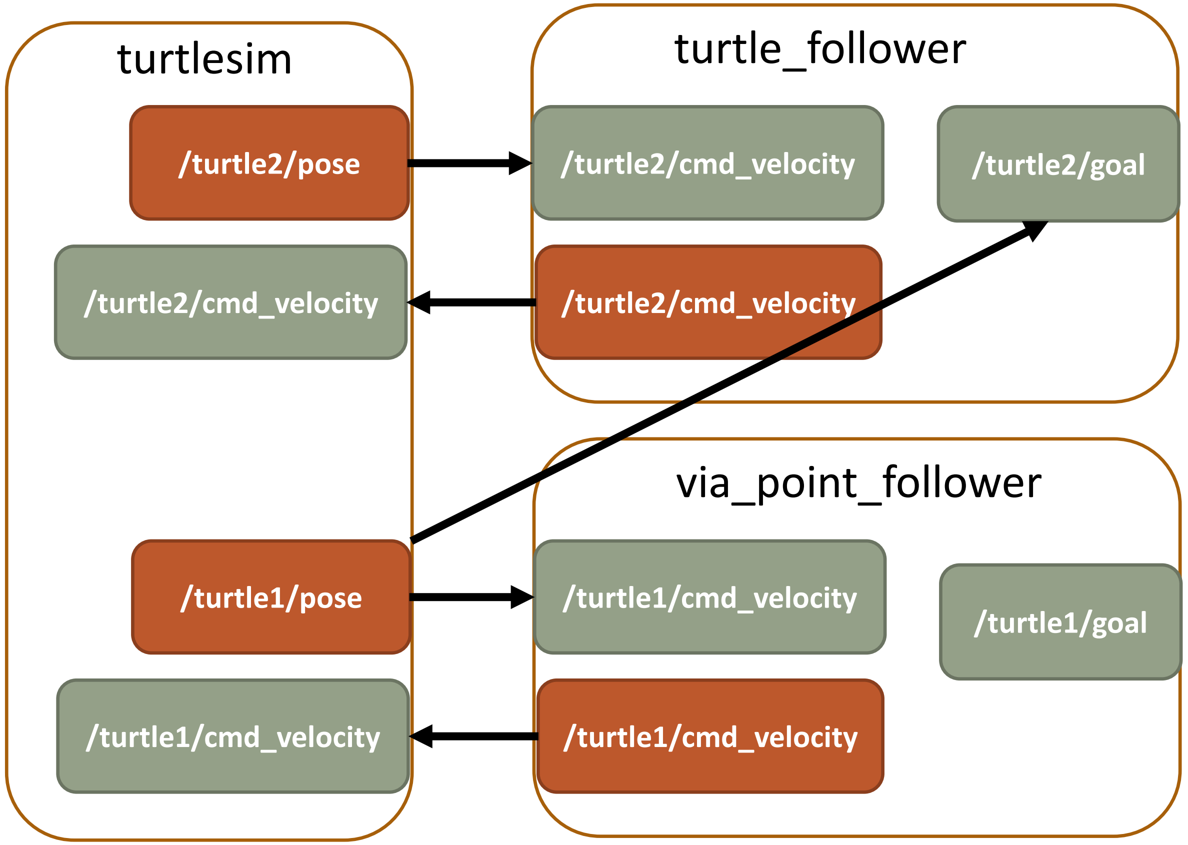
TurtleFollower

- subscribe to a turtle's pose

ViaPointFollower

- subscribe to a point

Exercise 2: Leader & Follower



Create 2 ROS nodes that allows a "follower" turtle to follow another "leader" turtle while the "leader" turtle follows a given goal.

These 2 nodes must share the same "parent" class.

Exercise 2: Leader & Follower

Terminal 1

```
>> ros2 run turtlesim turtlesim_node
```

Terminal 2

```
>> ros2 service call /spawn turtlesim/srv/Spawn "{x: 2.0, y: 2.0, theta: 0.0, name: 'turtle2'"}
```

Terminal 3

```
>> ros2 run turtlesim_control turtle_follower.py
```

Terminal 4

```
>> ros2 run turtlesim_control via_point_follower.py --ros-args -r /turtle2/goal:=/turtle1/pose
```

Exercise 2: Leader & Follower

Terminal 1

```
>> ros2 run turtlesim turtlesim_node
```

Terminal 2

```
>> ros2 service call /spawn turtlesim/srv/Spawn "{x: 2.0, y: 2.0, theta: 0.0, name: 'turtle2'"}
```

Terminal 3

```
>> ros2 run turtlesim_control turtle_follower.py
```

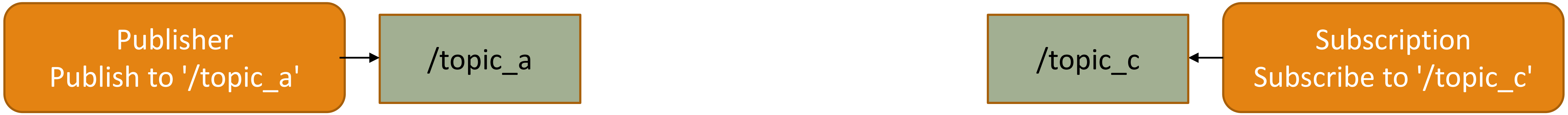
Terminal 4

```
>> ros2 run turtlesim_control via_point_follower.py --ros-args -r /turtle2/goal:=/turtle1/pose
```

What if the turtle's name changes to something else ?
Do we need to write a new file ?

Remapping & namespace

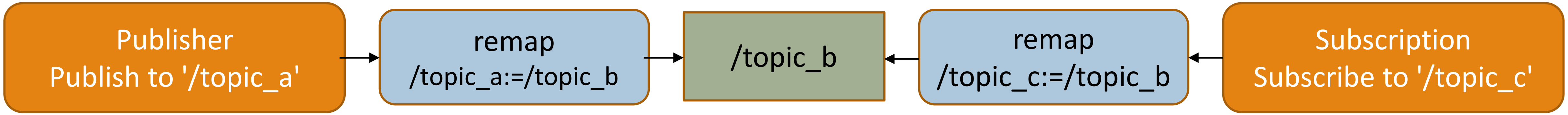
Inconsistent topic names



Remapping Topics

Outside of our code, we can "remap" the name of the topic in the command line.

In the code, we can change the subscribed topic to "pose" instead of using "/turtle1/pose".
We can apply the same idea to other topics.

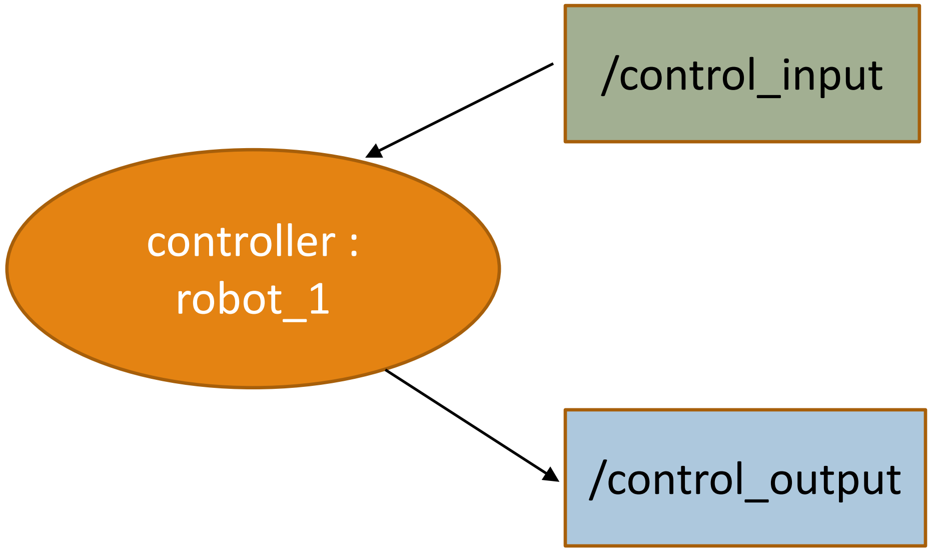


In the command line, we can add arguments at the end.

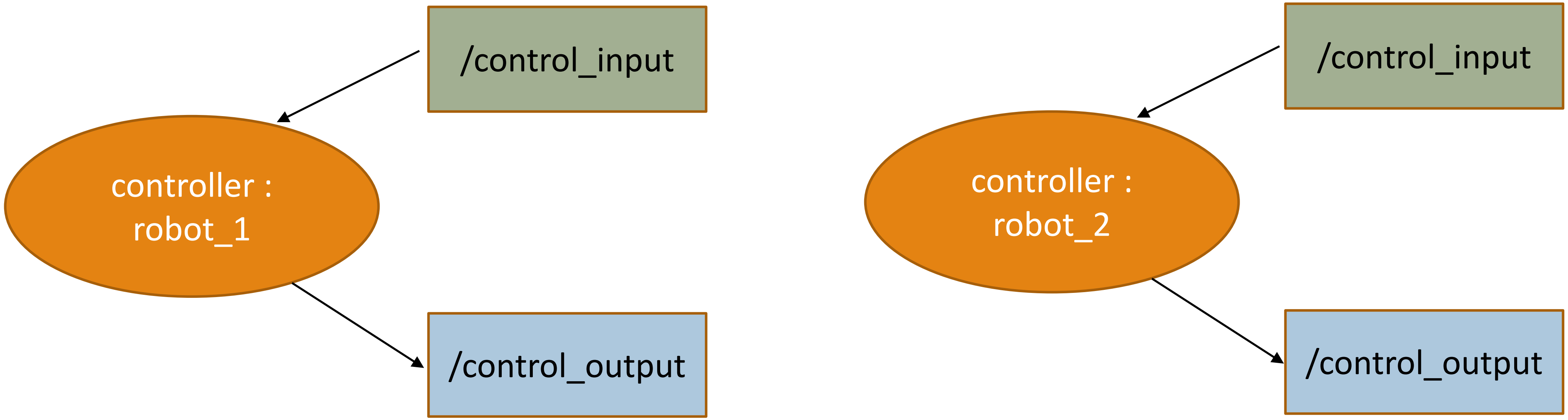
This only change the name, not the type

```
>> ros2 run turtlesim_control - -ros-args -r /pose:=/turtle1/pose
```

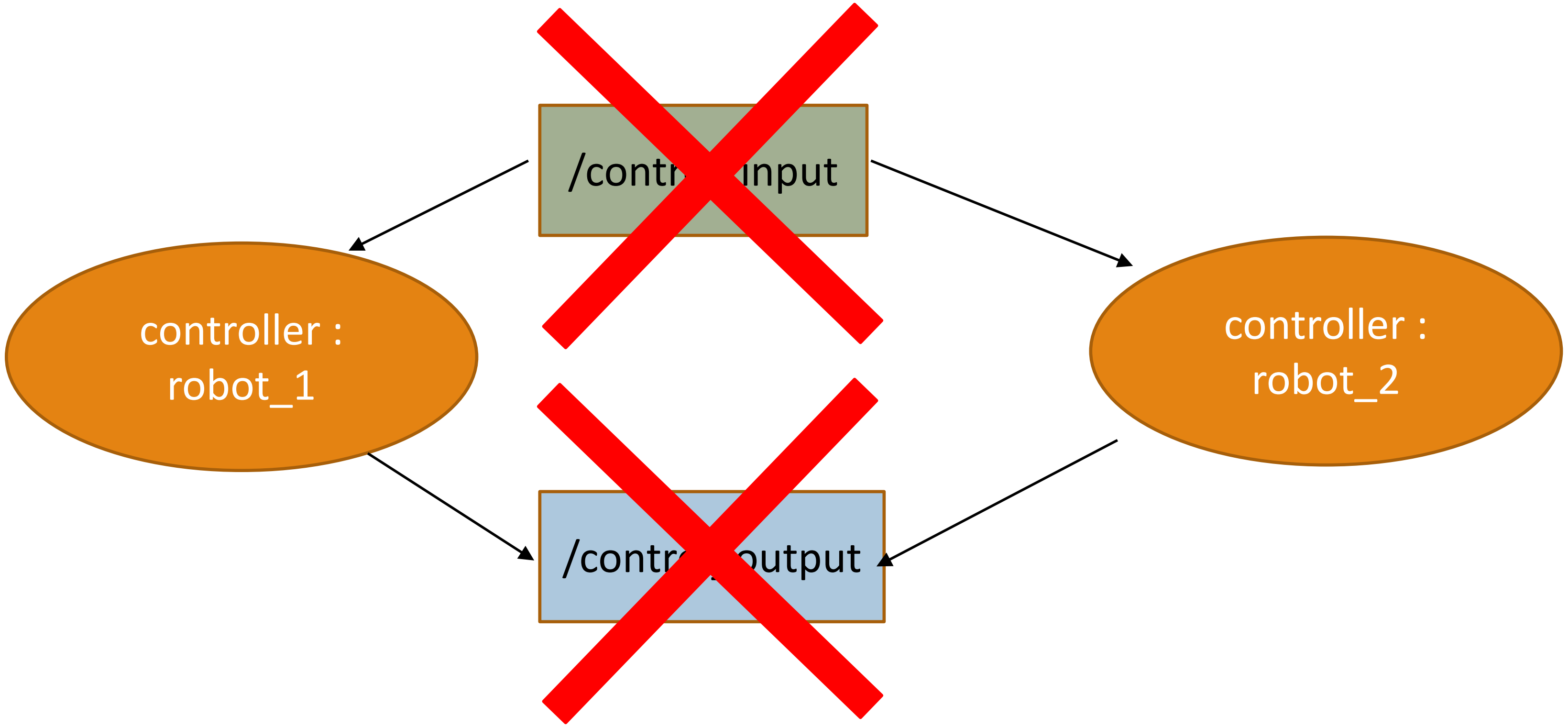
Having the same type of nodes in ROS network



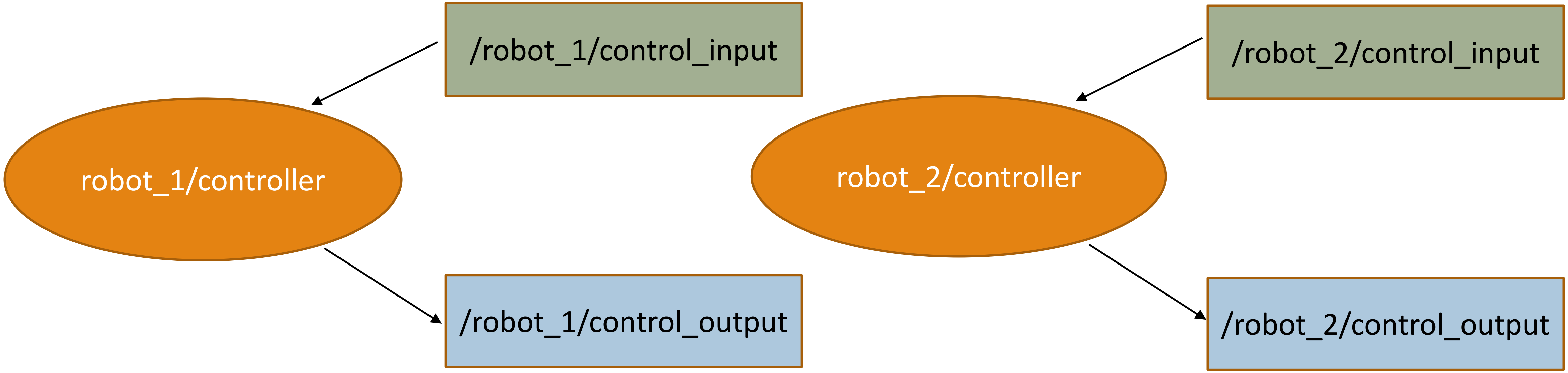
Having the same type of nodes in ROS network



Having the same type of nodes in ROS network



Distinguishing topics using namespace



Topic with Namespace

One can also add "namespace" to an entire node, which appends the namespace to the front of every name without "/" in the front.

For example,

pose => /namespace/pose
turtle_follower = > /namespace/turtle_follower
/goal => /goal

In the command line, we can add arguments at the end.

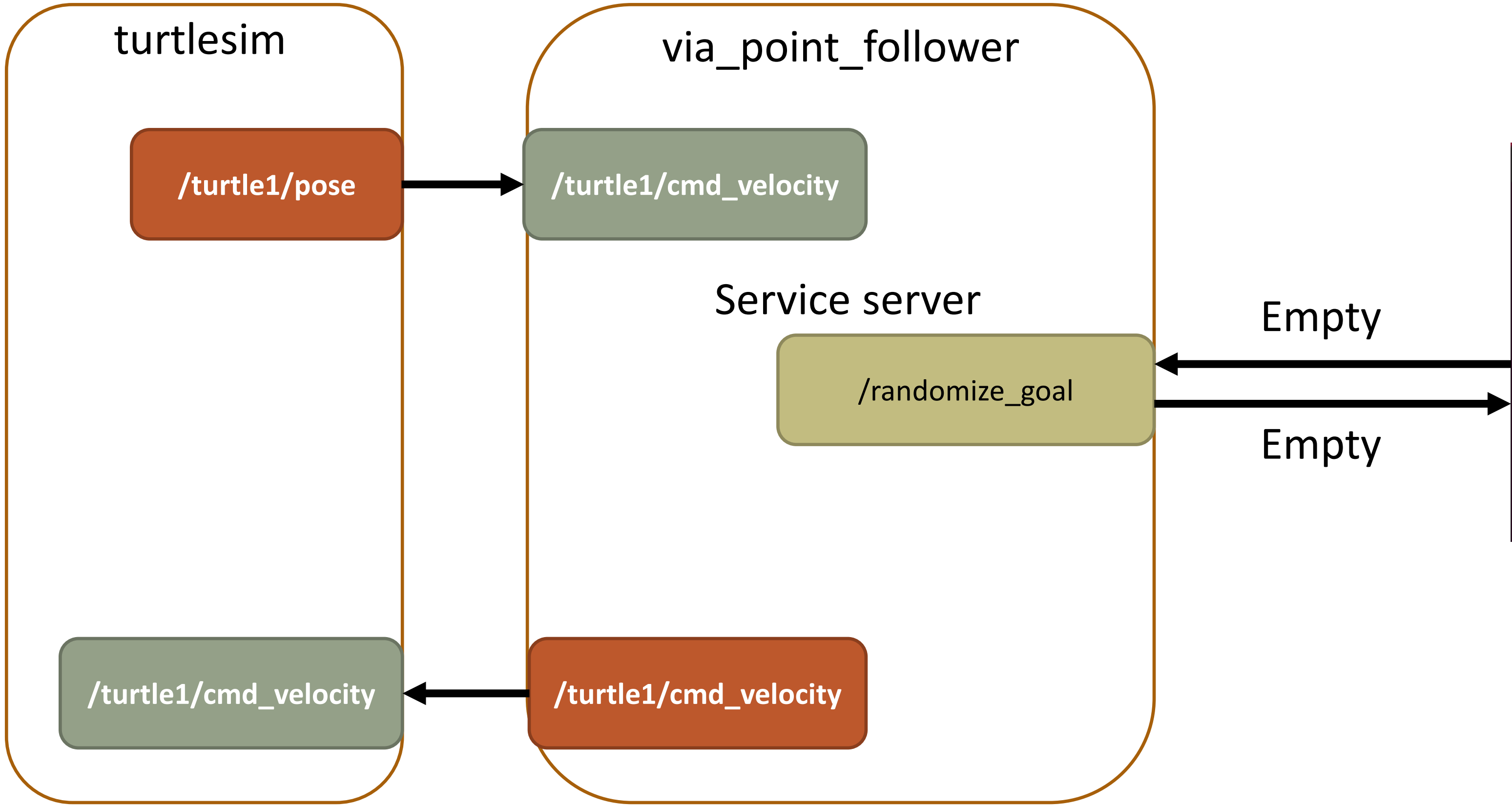
```
>> ros2 run turtlesim_control - -ros-args -r __ns:=/turtle1
```

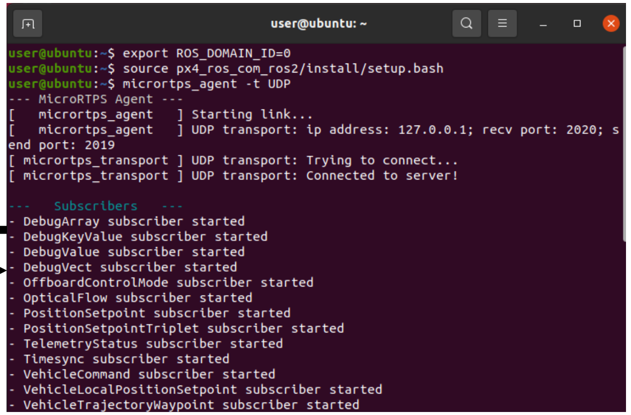
Service Server & Service Client

Attaching a service server and its callback

```
from std_srvs/srv import Empty

class NewNode(Node):
    def __init__(self):
        super().__init__('node_name')
        self.count = 0
        self.srv = self.create_service(Empty,'my_service',self.srv_callback)
    def srv_callback(self,request,response):
        self.count = self.count + 1
        return response
```

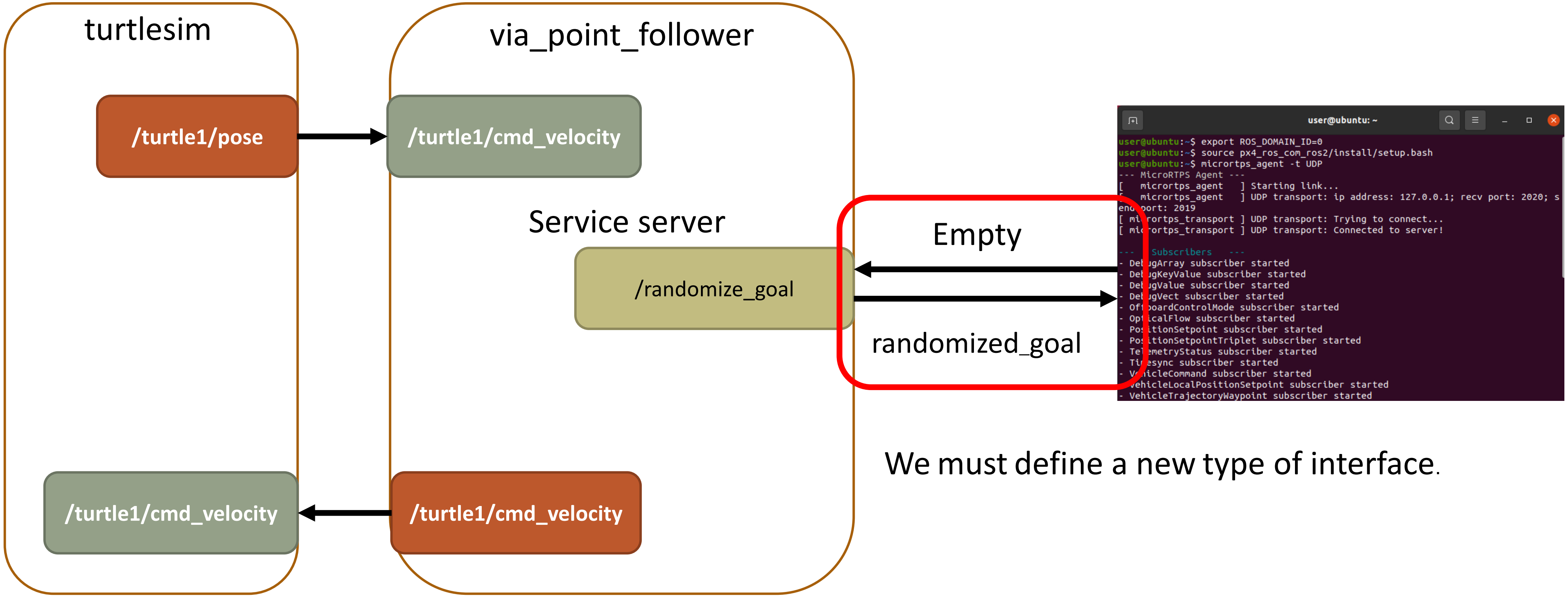
Exercise 3: Controller with services





```
user@ubuntu:~$ export ROS_DOMAIN_ID=0
user@ubuntu:~$ source px4_ros_com_ros2/install/setup.bash
user@ubuntu:~$ micrortps_agent -t UDP
--- MicroRTPS Agent ---
[   micrortps_agent   ] Starting link...
[   micrortps_agent   ] UDP transport: ip address: 127.0.0.1; recv port: 2020; s
end port: 2019
[ micrortps_transport ] UDP transport: Trying to connect...
[ micrortps_transport ] UDP transport: Connected to server!

---   Subscribers   ---
- DebugArray subscriber started
- DebugKeyValue subscriber started
- DebugValue subscriber started
- DebugVect subscriber started
- OffboardControlMode subscriber started
- OpticalFlow subscriber started
- PositionSetpoint subscriber started
- PositionSetpointTriplet subscriber started
- TelemetryStatus subscriber started
- Timesync subscriber started
- VehicleCommand subscriber started
- VehicleLocalPositionSetpoint subscriber started
- VehicleTrajectoryWaypoint subscriber started
```

Exercise 3: Controller with services



```
user@ubuntu:~$ export ROS_DOMAIN_ID=0
user@ubuntu:~$ source px4_ros_com_ros2/install/setup.bash
user@ubuntu:~$ micrortps_agent -t UDP
--- MicroRTPS Agent ---
[   micrortps_agent   ] Starting link...
[   micrortps_agent   ] UDP transport: ip address: 127.0.0.1; recv port: 2020; s
end port: 2019
[ micrortps_transport ] UDP transport: Trying to connect...
[ micrortps_transport ] UDP transport: Connected to server!

---   Subscribers   ---
- DebugArray subscriber started
- DebugKeyValue subscriber started
- DebugValue subscriber started
- DebugVect subscriber started
- OffboardControlMode subscriber started
- OpticalFlow subscriber started
- PositionSetpoint subscriber started
- PositionSetpointTriplet subscriber started
- TelemetryStatus subscriber started
- Timesync subscriber started
- VehicleCommand subscriber started
- VehicleLocalPositionSetpoint subscriber started
- VehicleTrajectoryWaypoint subscriber started
```

We must define a new type of interface.

Custom Interface

Pose.msg

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

InvKin.srv

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
---
sensor_msgs/JoinState joint_config
```

.action

```
geometry_msgs/Point position
geometry_msgs/Quaternion
orientation
---
sensor_msgs/JoinState joint_config
---
int64 sec
int64 nanosec
```

Interface name must start with uppercase letter and must not contain underscore or other unqie characters. The field itself must start with lowercase letter.

Custom Interface

Pose.msg

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

InvKin.srv

request message

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
---
sensor_msgs/JoinState joint_config
```

response message

.action

```
geometry_msgs/Point position
geometry_msgs/Quaternion
orientation
---
sensor_msgs/JoinState joint_config
---
int64 sec
int64 nanosec
```

Custom Interface

Pose.msg

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

InvKin.srv

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
---
sensor_msgs/JoinState joint_config
```

.action

goal message

```
geometry_msgs/Point position
geometry_msgs/Quaternion
orientation
```

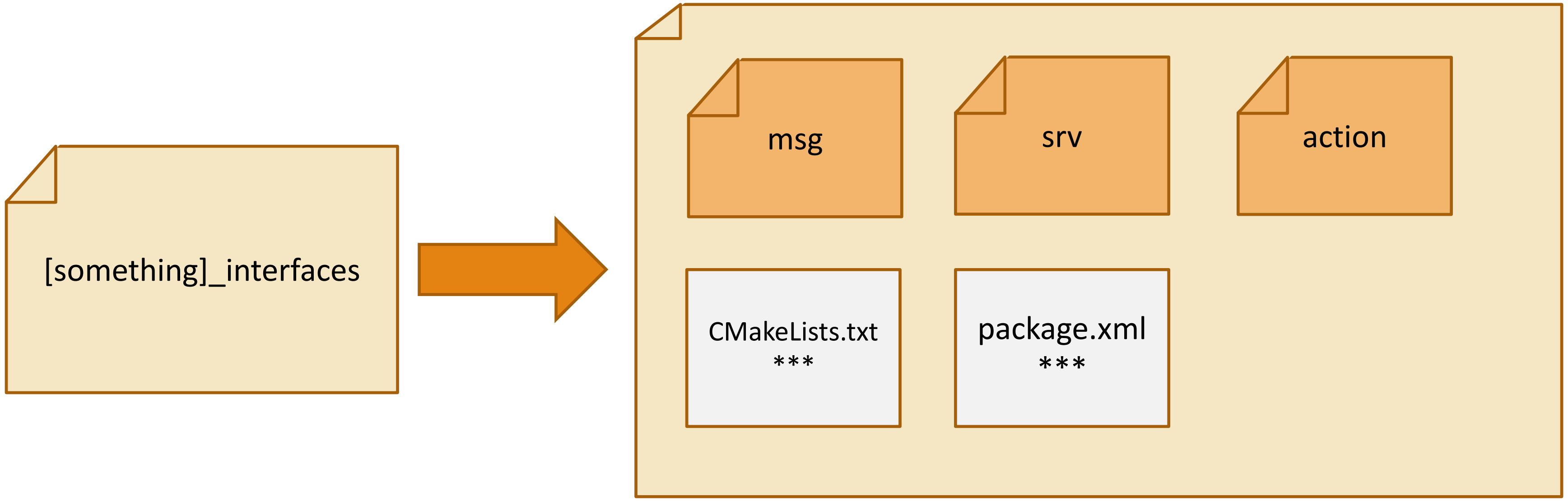
result message

```
---
sensor_msgs/JoinState joint_config
```

```
int64 sec
int64 nanosec
```

feedback message

Package with custom interface



CMakeLists.txt (custom interface)

Search for dependent packages

Generating custom definition

Associating the dependency

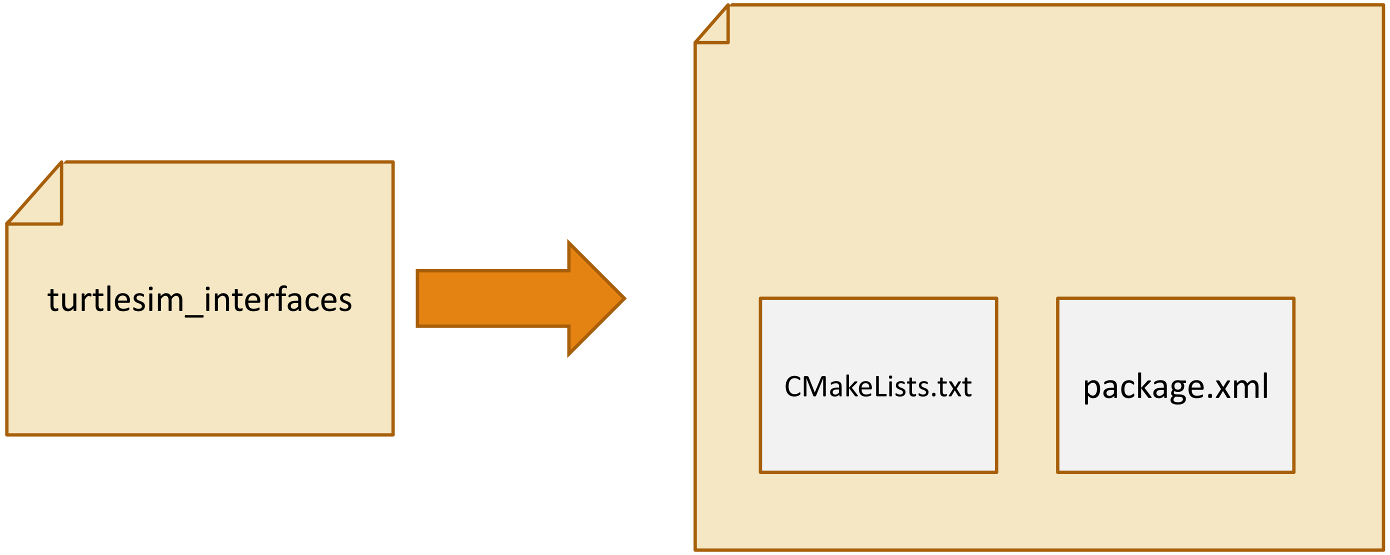
```
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Num.msg"
  "msg/Sphere.msg"
  "srv/AddThreeInts.srv"
  DEPENDENCIES geometry_msgs
)
```

package.xml (custom interface)

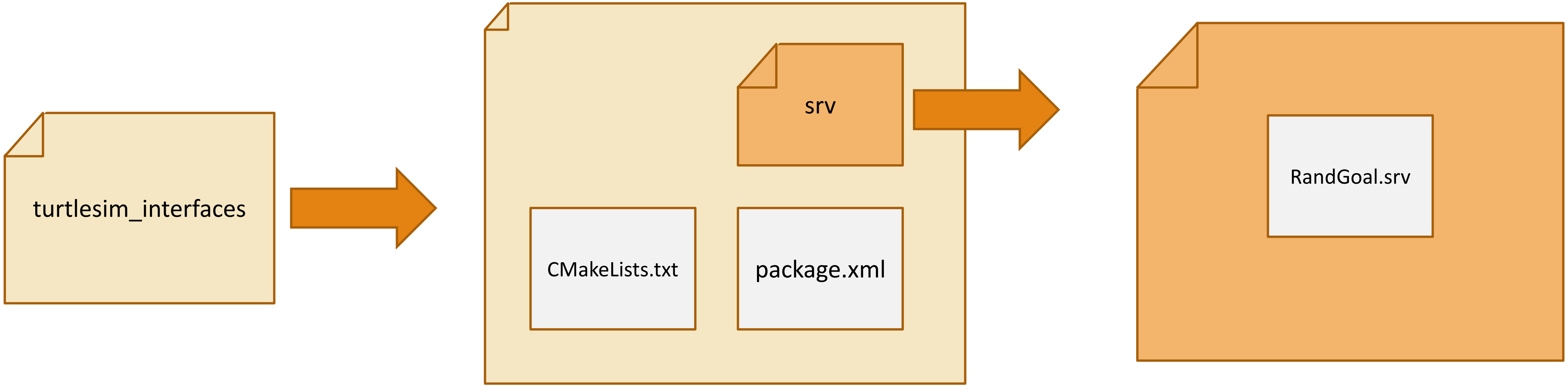
Adding dependent packages

```
<depend>geometry_msgs</depend>
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

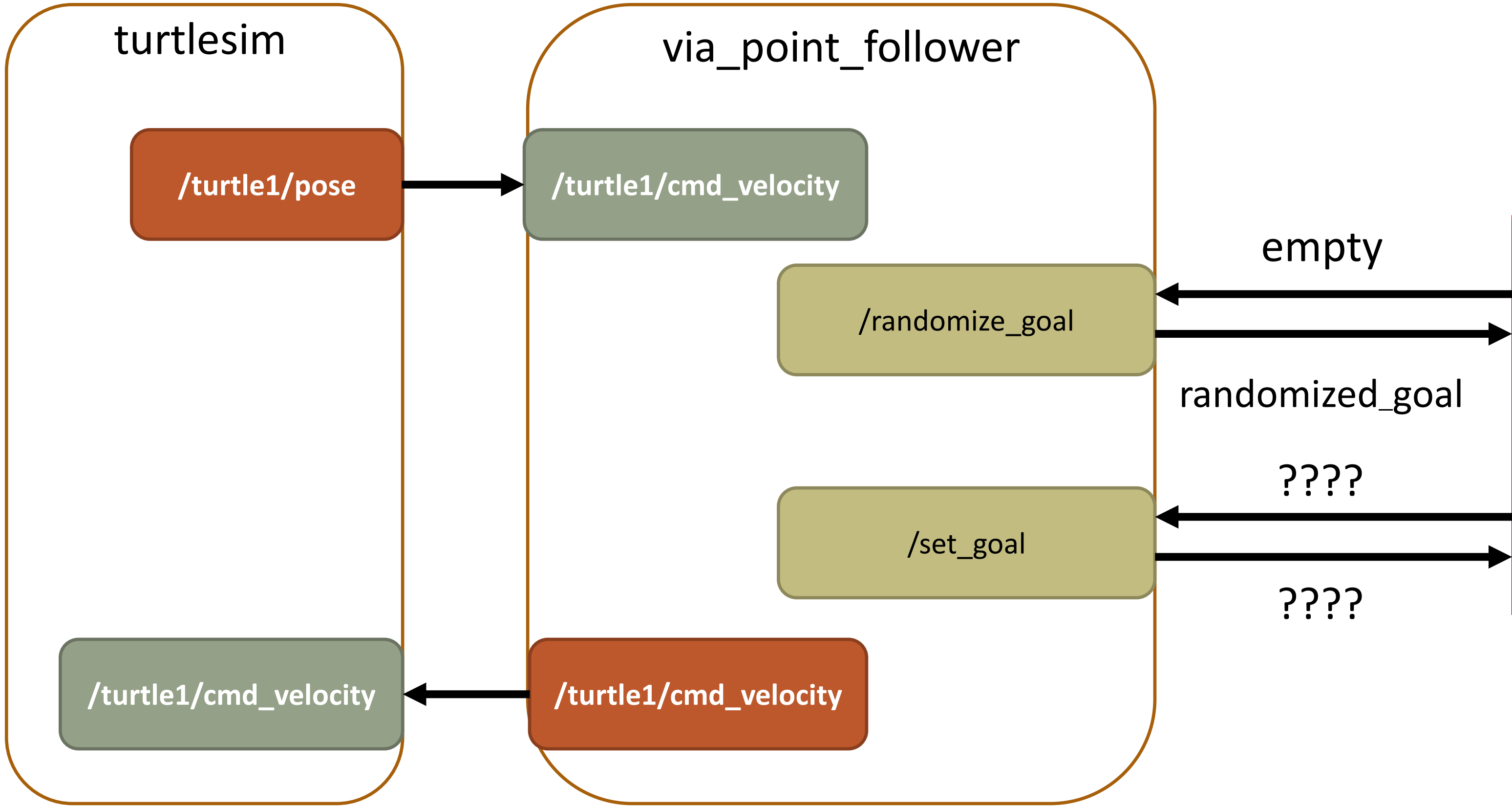
Custom Interface for our turtlesim

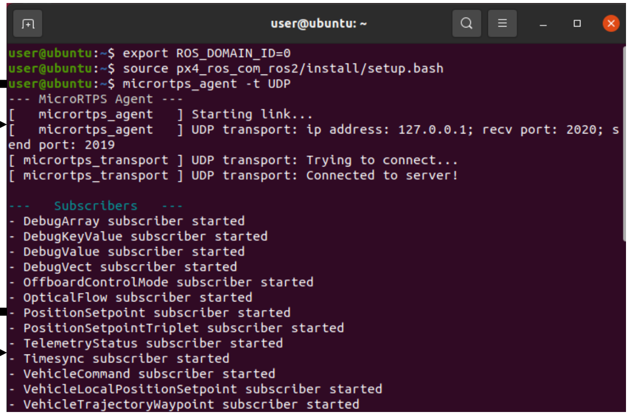


custom service



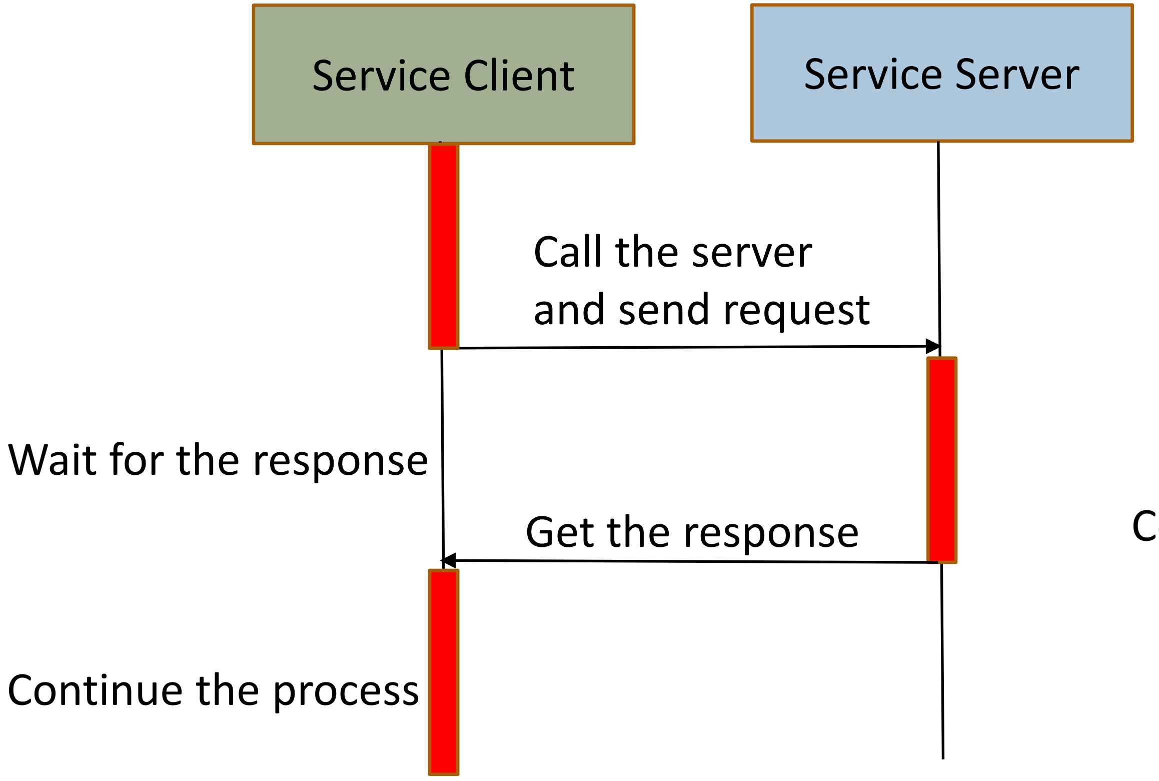
Exercise 3: Controller with services



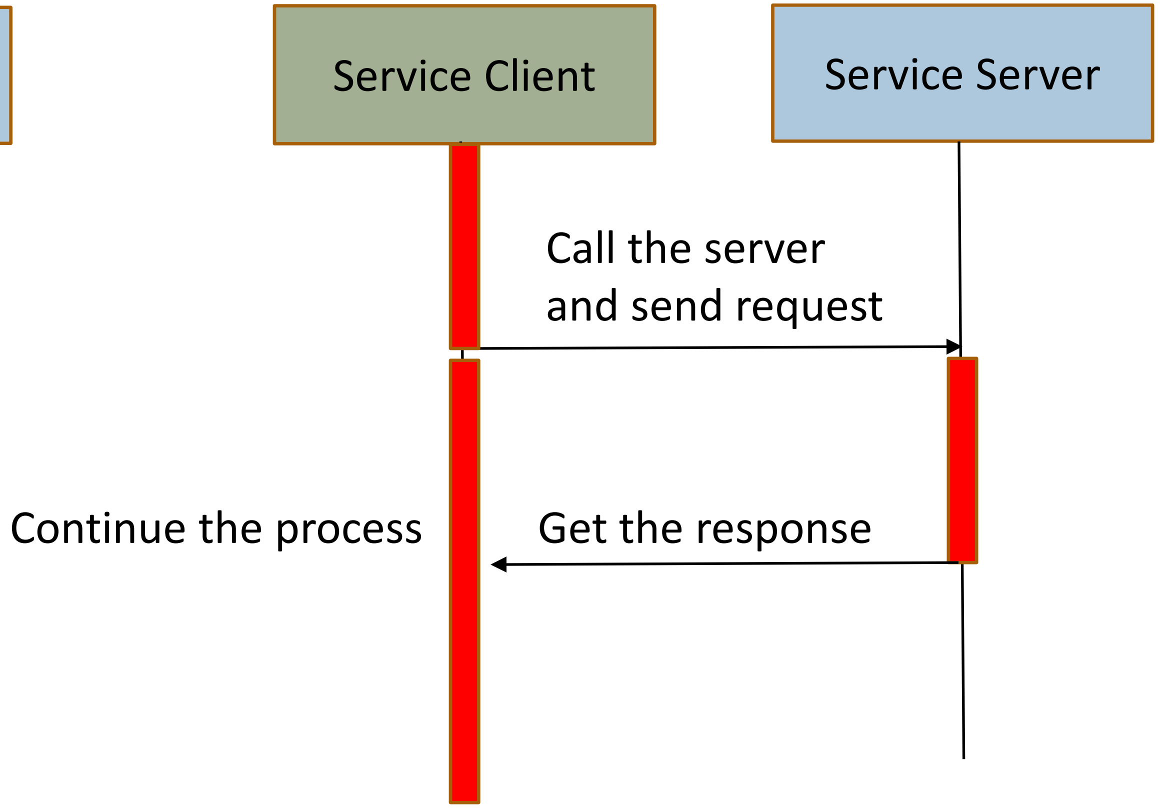


```
user@ubuntu:~$ export ROS_DOMAIN_ID=0
user@ubuntu:~$ source px4_ros_com_ros2/install/setup.bash
user@ubuntu:~$ micrortps_agent -t UDP
--- MicroRTPS Agent ---
[   micrortps_agent   ] Starting link...
[   micrortps_agent   ] UDP transport: ip address: 127.0.0.1; recv port: 2020; s
end port: 2019
[ micrortps_transport ] UDP transport: Trying to connect...
[ micrortps_transport ] UDP transport: Connected to server!

---   Subscribers   ---
- DebugArray subscriber started
- DebugKeyValue subscriber started
- DebugValue subscriber started
- DebugVect subscriber started
- OffboardControlMode subscriber started
- OpticalFlow subscriber started
- PositionSetpoint subscriber started
- PositionSetpointTriplet subscriber started
- TelemetryStatus subscriber started
- Timesync subscriber started
- VehicleCommand subscriber started
- VehicleLocalPositionSetpoint subscriber started
- VehicleTrajectoryWaypoint subscriber started
```

Synchronous vs Asynchronous Programming

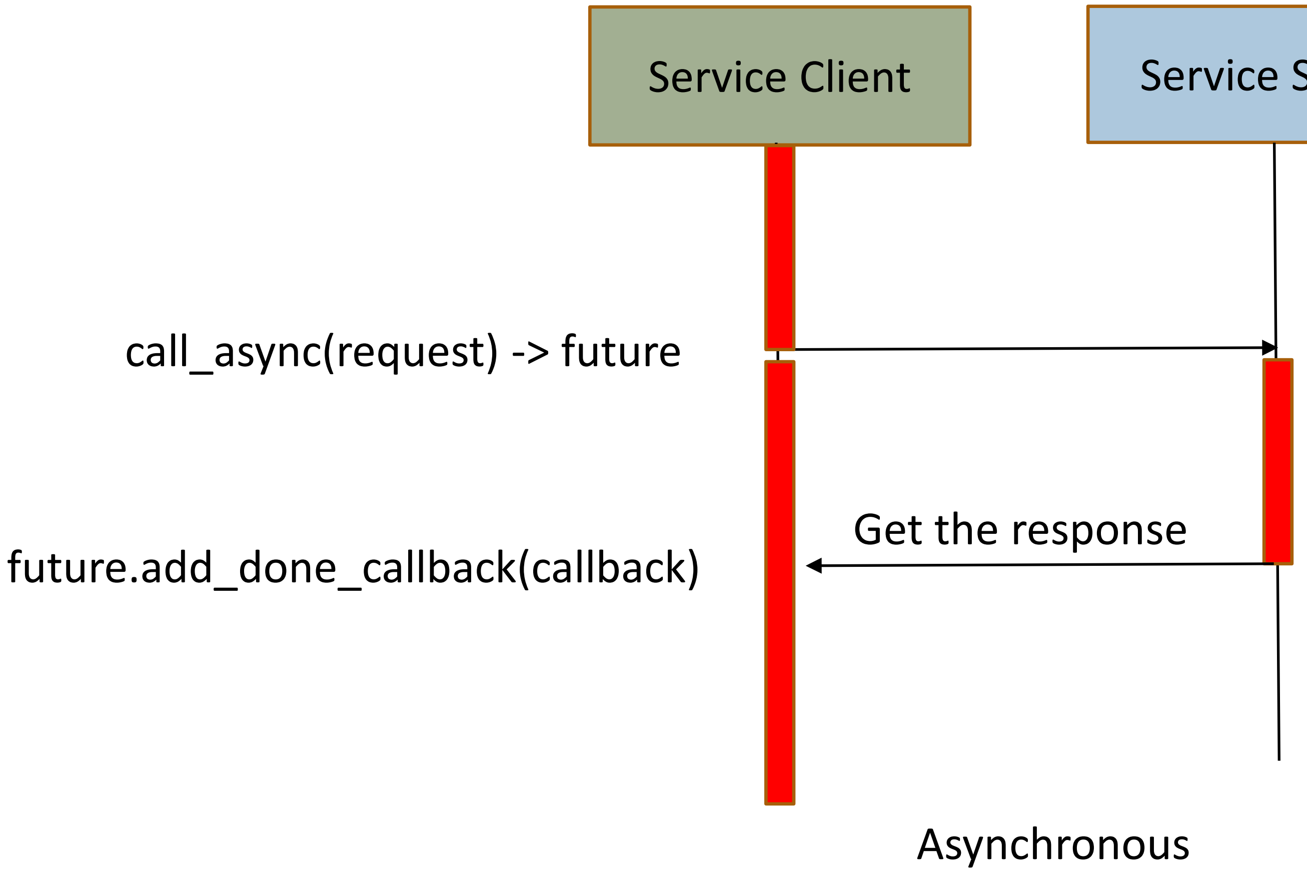


Synchronous



Asynchronous

Future Object (RCLPY)

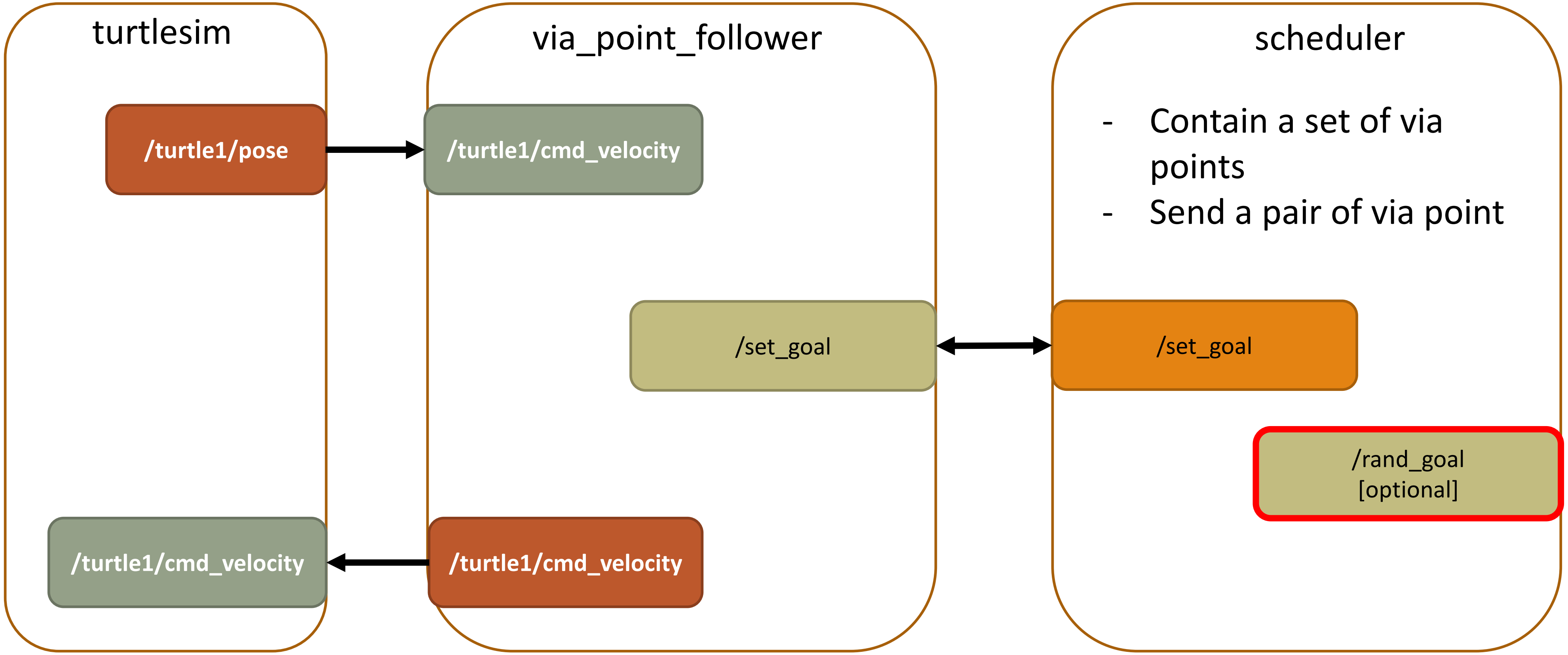

Service Client
Service Server
call_async(request) -> future
Get the response
future.add_done_callback(callback)
Asynchronous

Example:
If the "future" is done, print and update some variables.

Attaching a service client

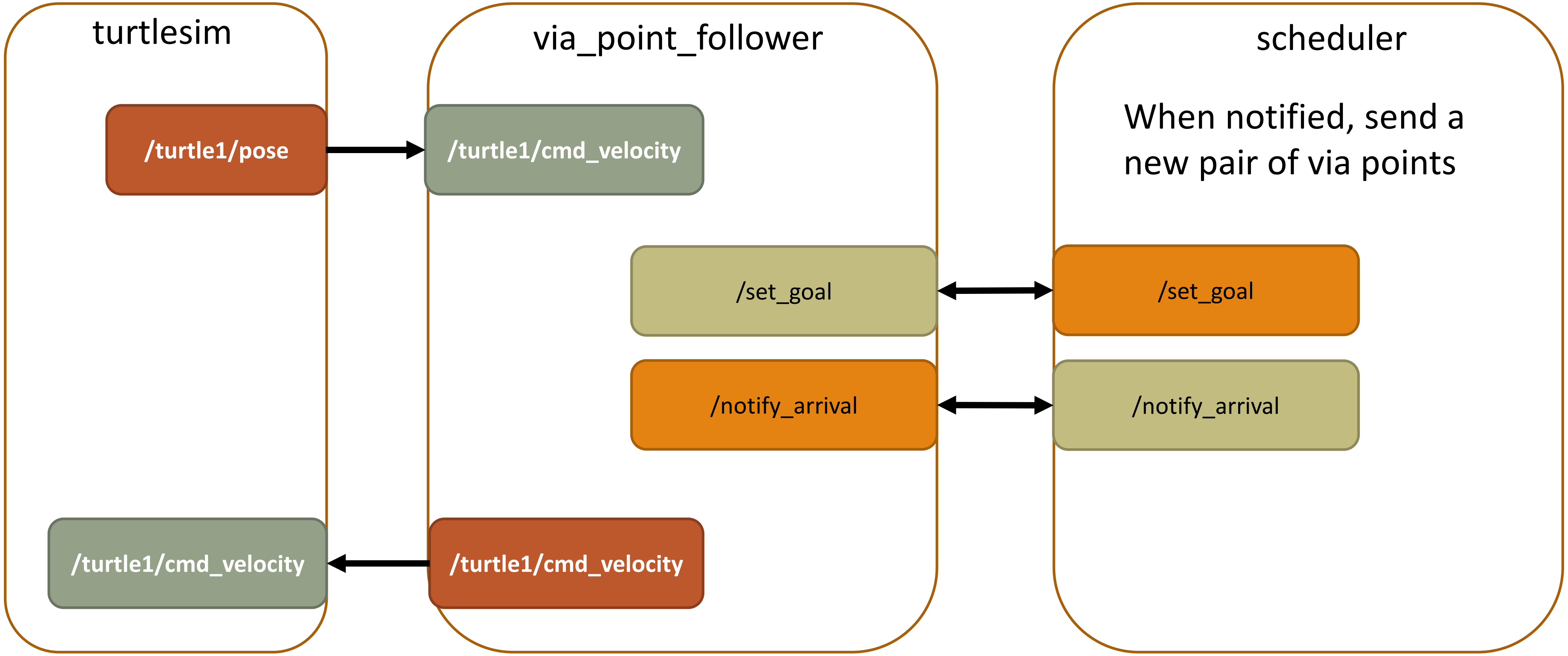
```
from std_srvs/srv import Empty

class NewNode(Node):
    def __init__(self):
        super().__init__('node_name')
        self.cli = self.create_client(Empty,'my_service')
        req = Empty.Request()
        self.future = self.cli.call_async(req)
        self.future.add_done_callback(self.done_callback)
    def done_callback(self,future):
        print(future.result())
```

Exercise 4: Controller with services



- Contain a set of via points
- Send a pair of via point

Exercise 4: Controller with services



When notified, send a new pair of via points

Exercise 4: Controller with services

Terminal 1

```
>> ros2 run turtlesim turtlesim_node
```

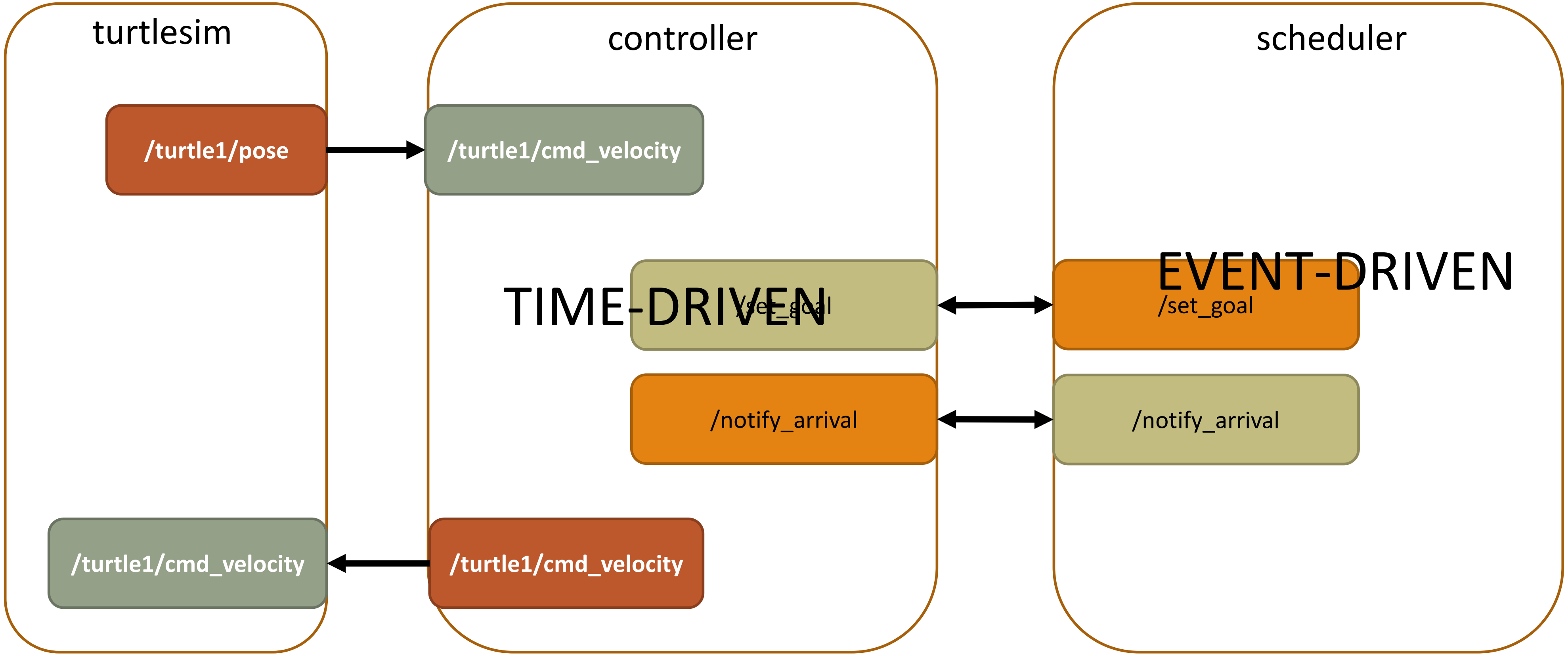
Terminal 2

```
>> ros2 run turtlesim_control via_point_follower.py --ros-args -r __ns:=/turtle1
```

Terminal 3

```
>> ros2 run turtlesim_control scheduler.py __ns:=/turtle1
```

Time-driven vs. Event-driven



Waiting for a task to complete

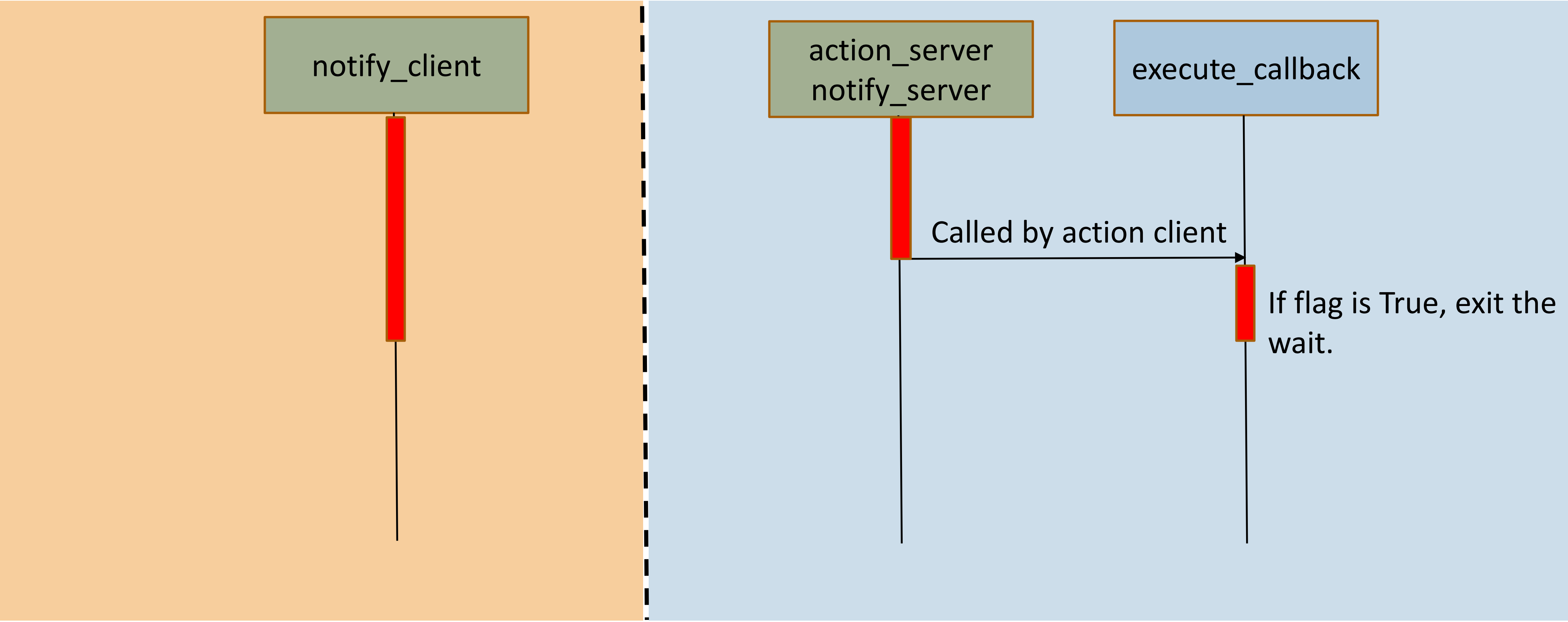
What if instead of only setting a goal, we also wait for the turtle to reach the goal ?

When it reaches the goal, it should return the total distance (Euclidean).

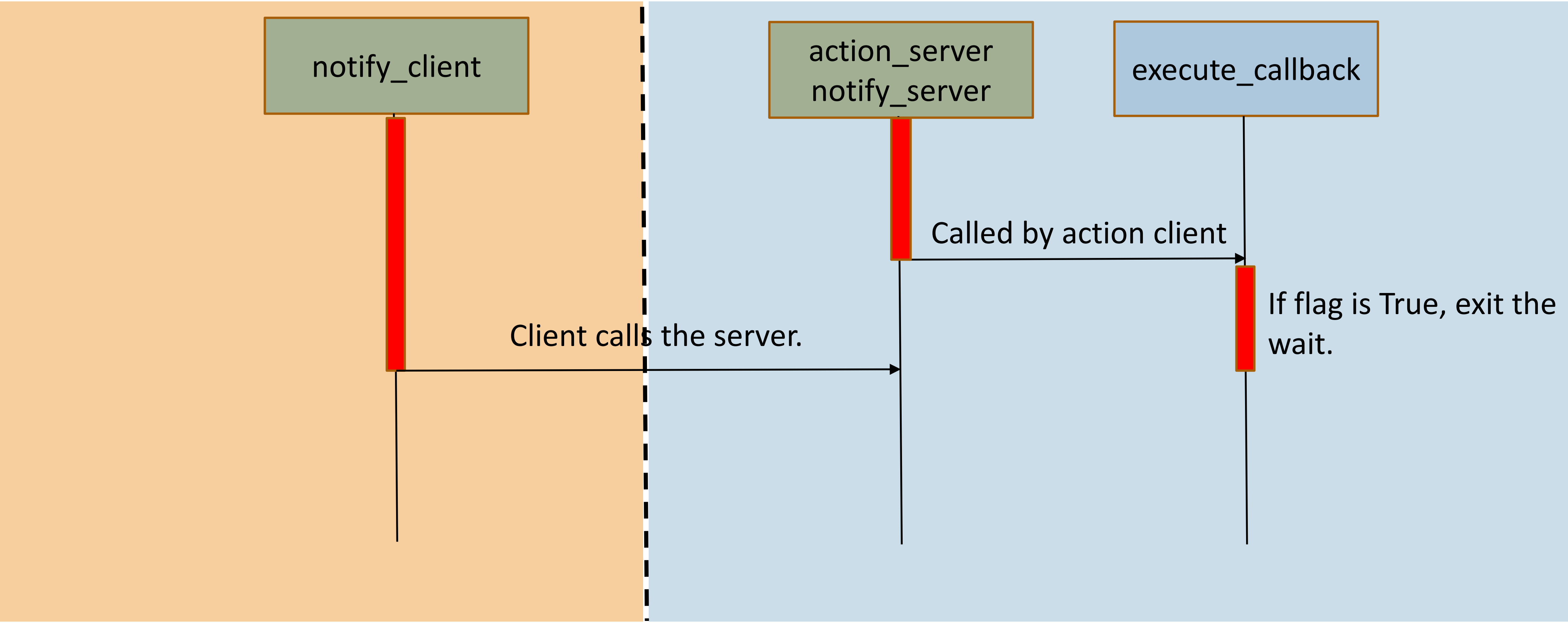
In the meantime, it should publish the elasped time.

Action, Callback Group, & Multithread Execution

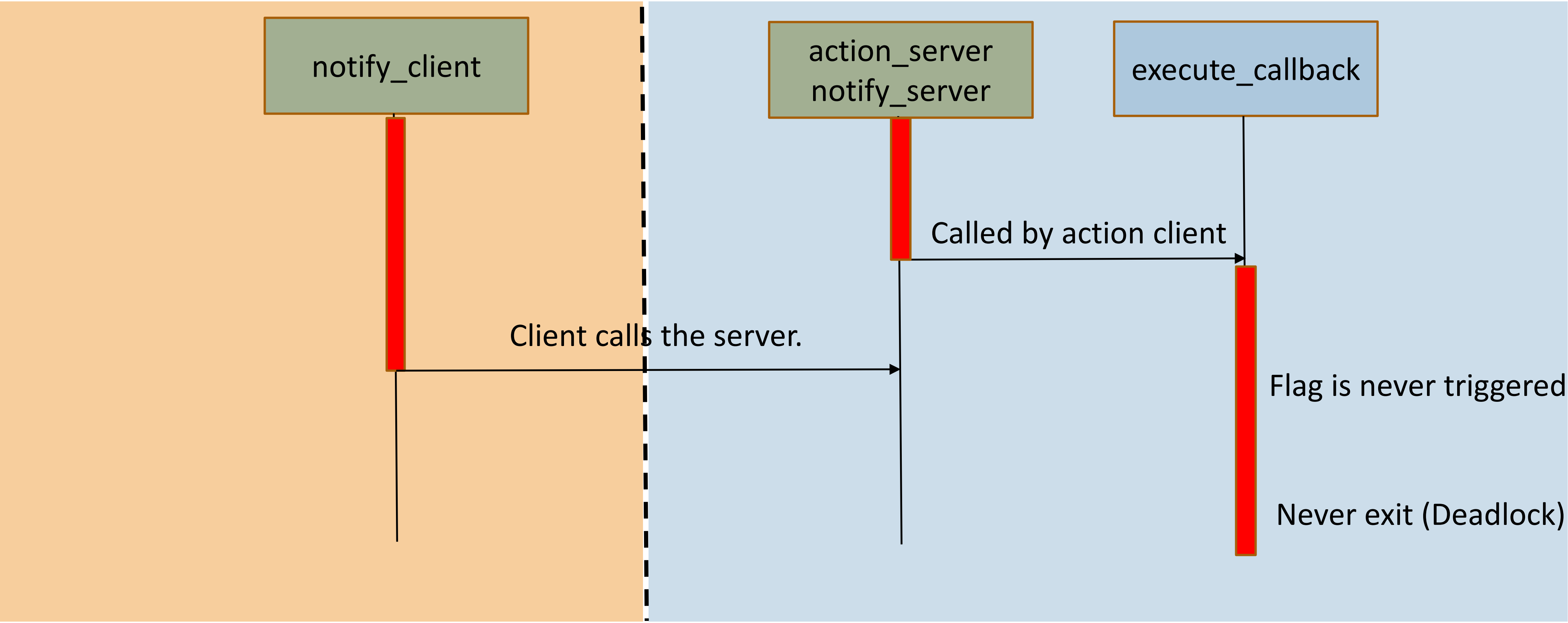
Action & Single Thread Execution



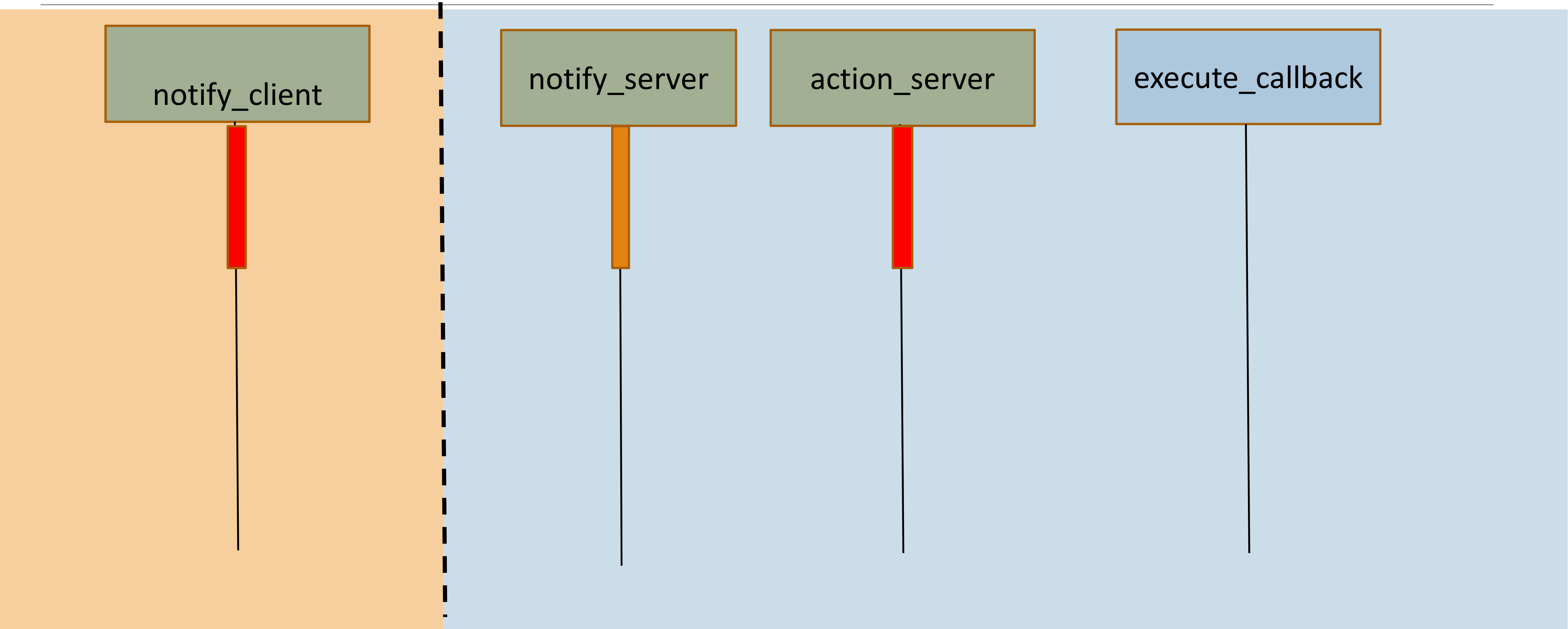
Action & Single Thread Execution



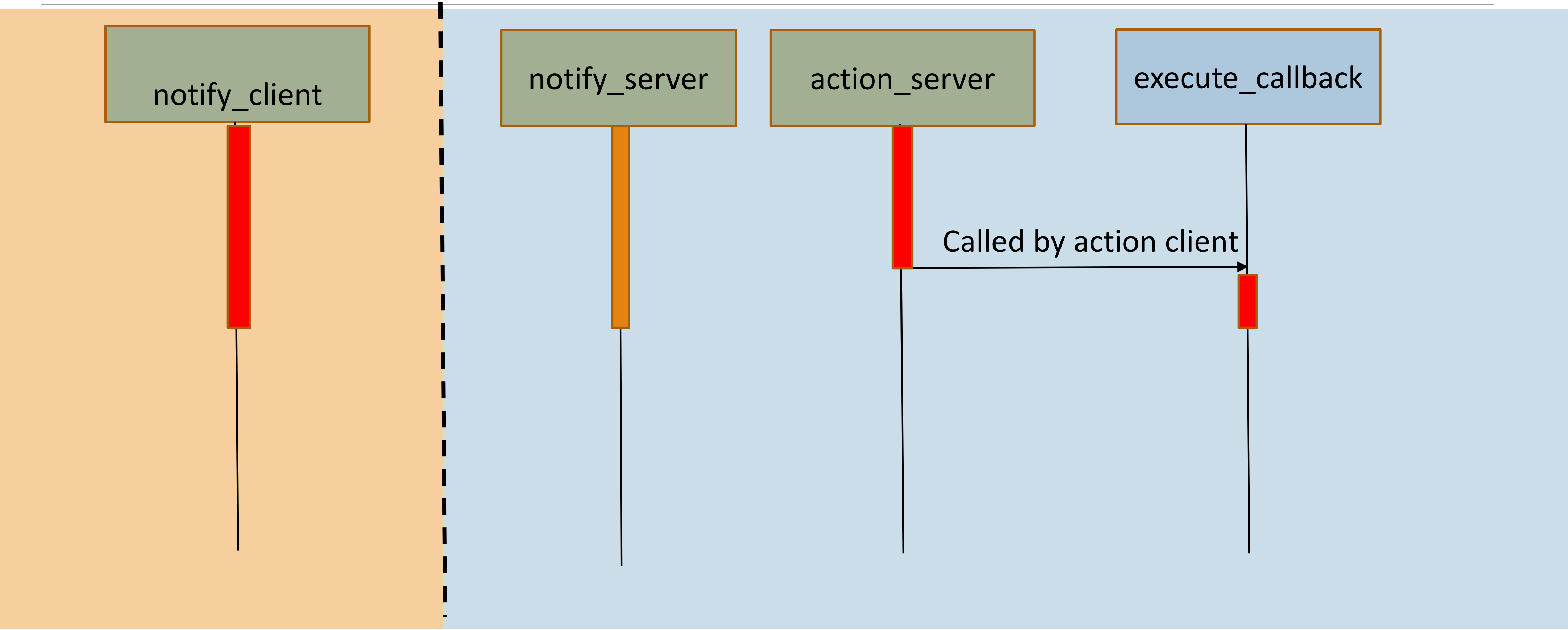
Deadlock


notify_client
action_server
notify_server
execute_callback
Called by action client
Client calls the server.
Flag is never triggered
Never exit (Deadlock)

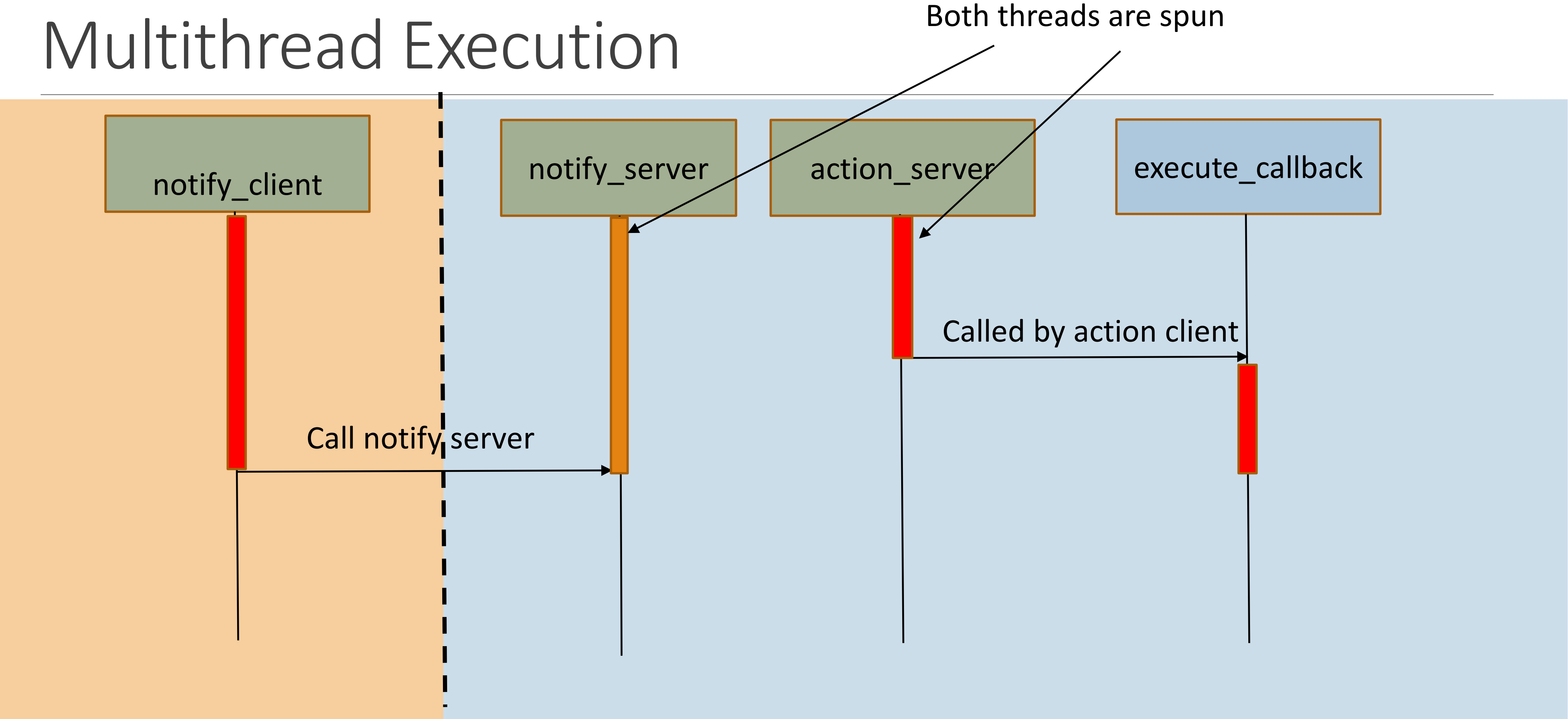
Multithread Execution



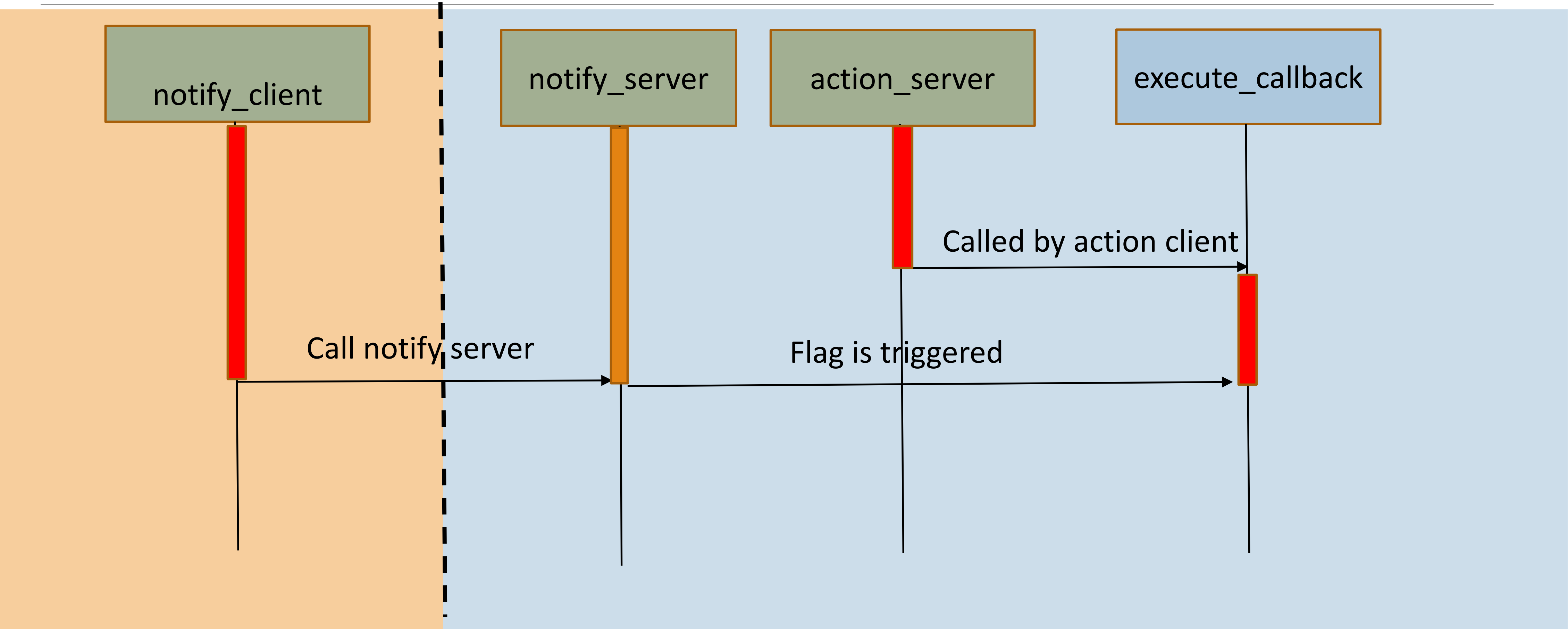
Multithread Execution



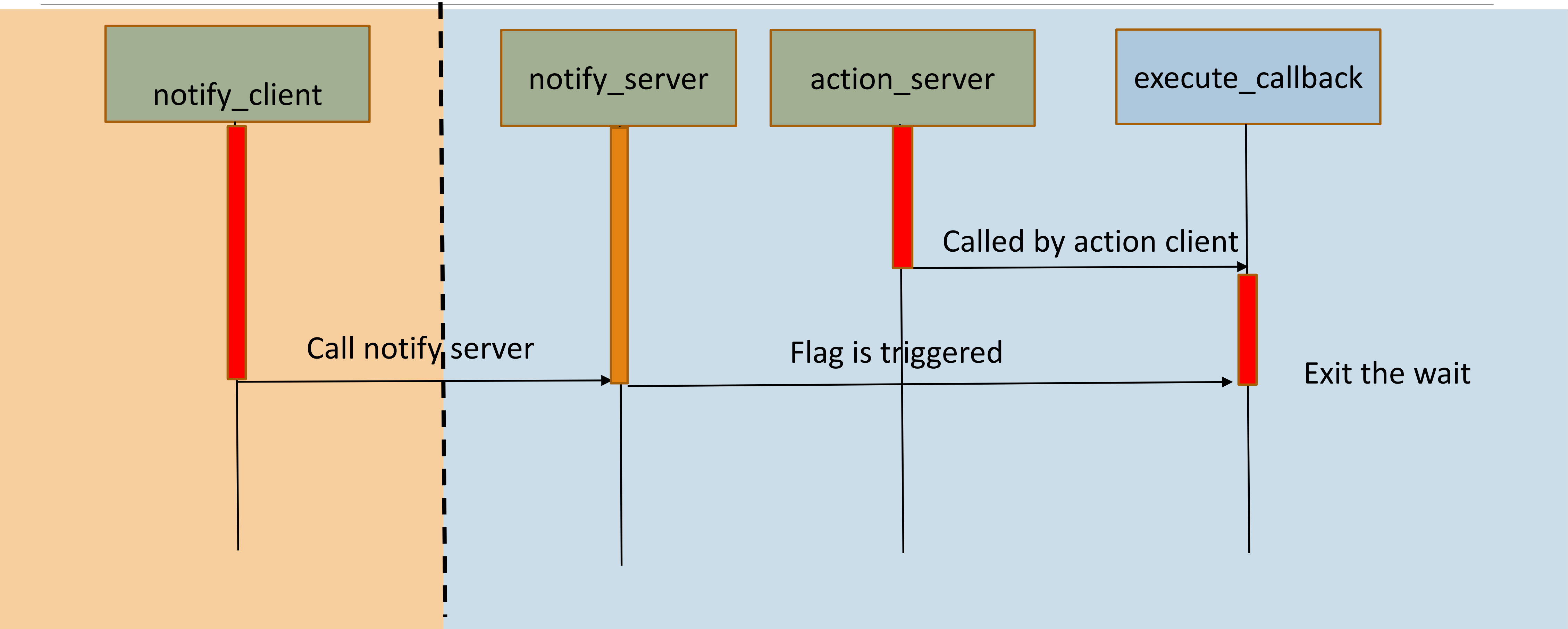
Multithread Execution



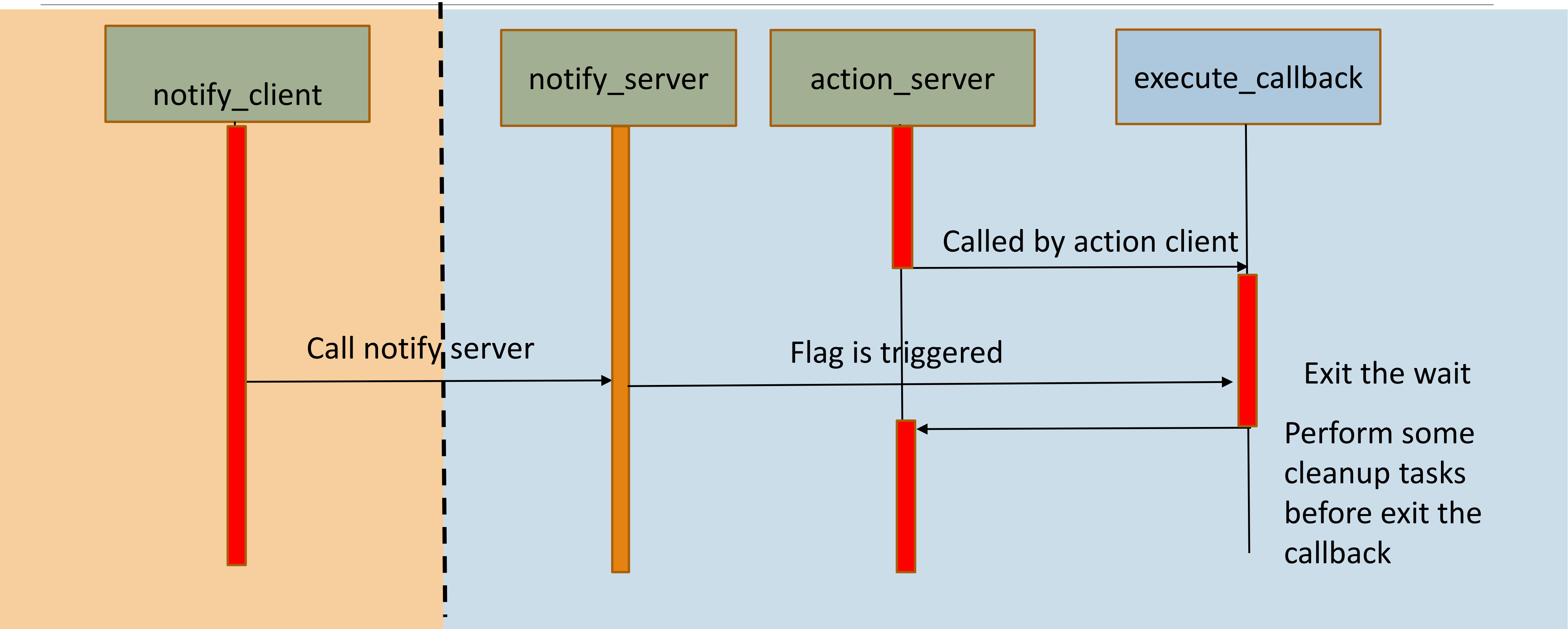
Multithread Execution



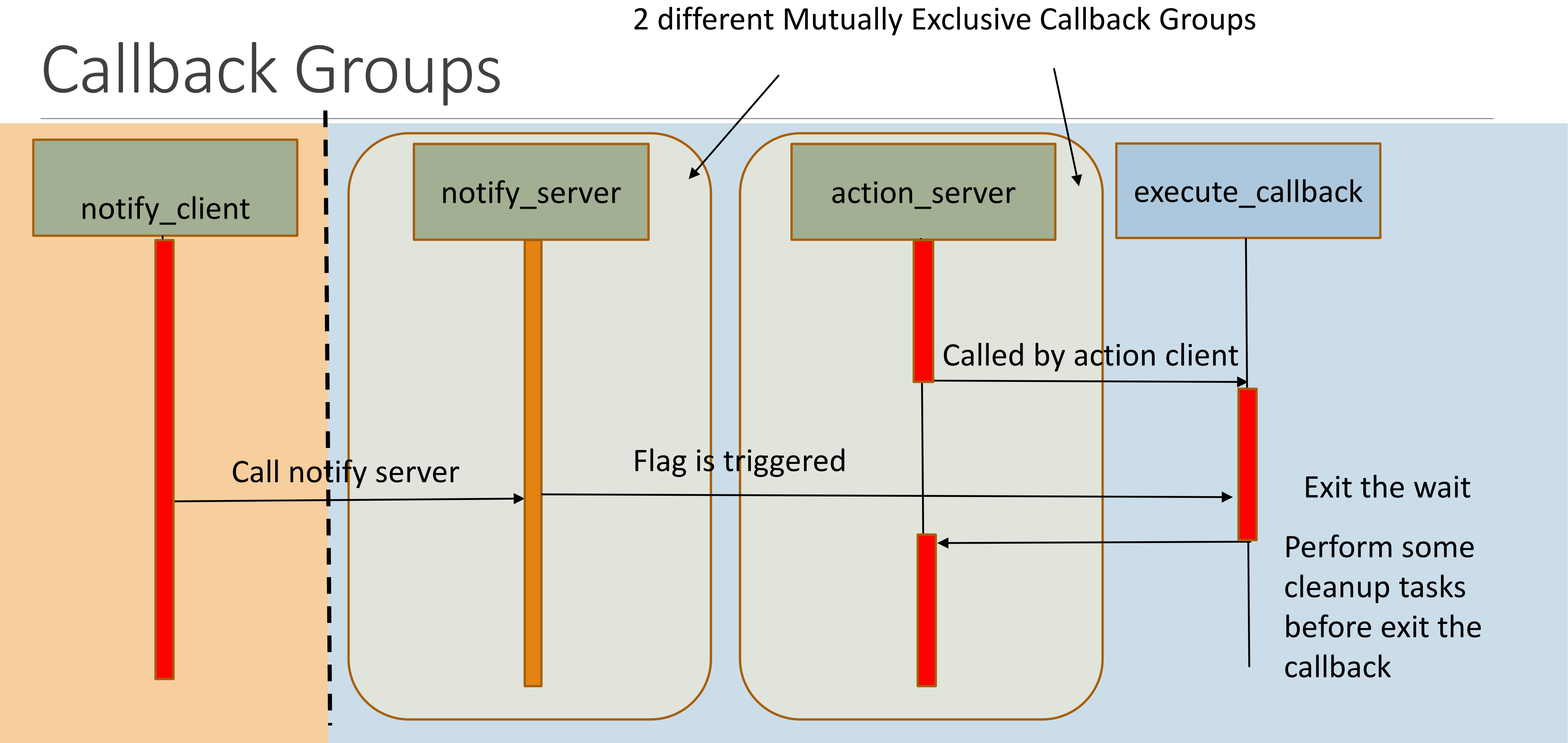
Multithread Execution



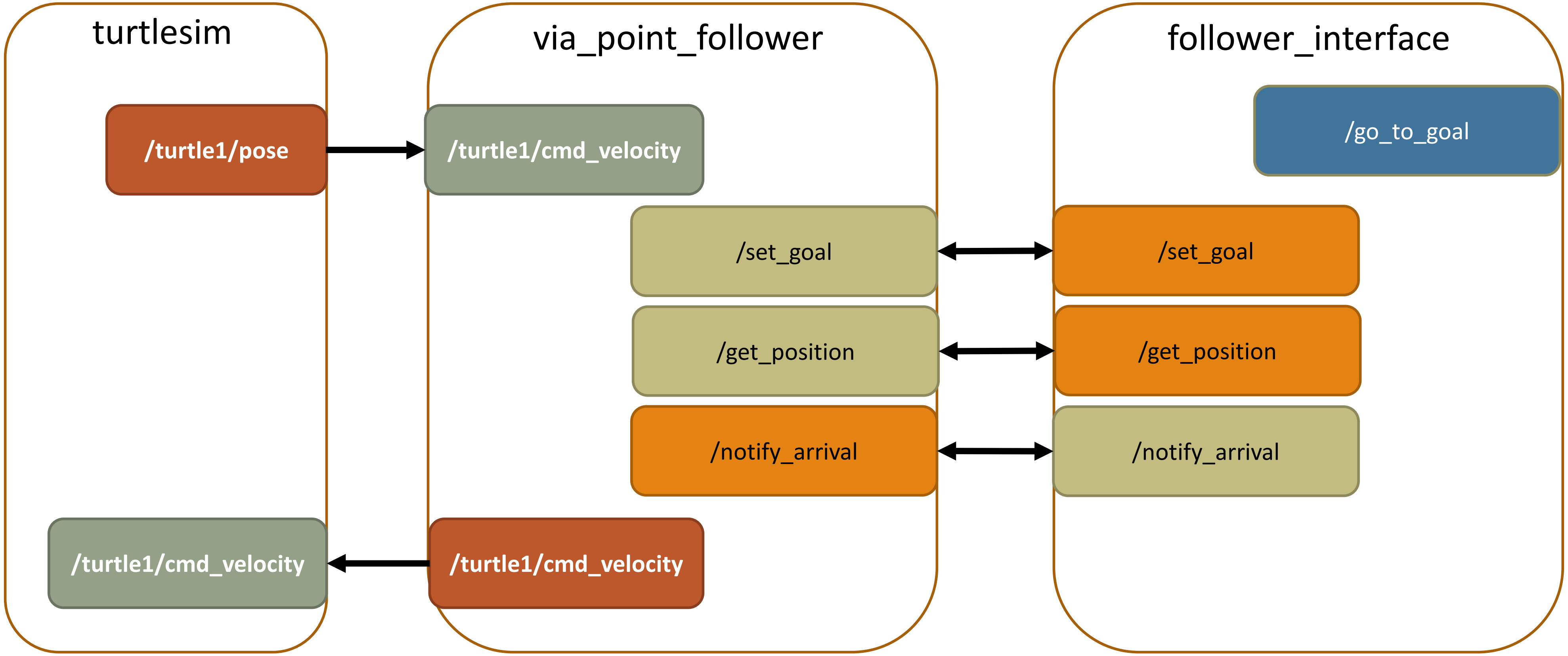
Multithread Execution



Callback Groups



Exercise 5: Action Interface



Summary

- History of ROS
- Getting to know ROS
- Introduction to ROS 2
- Navigating Linux Terminal
- Using ROS2 in Terminal
- ROS2 Node Programming with RCLPY
- OOP with RCPLY
- Custom Interface, Service Server, Service Client
- Action, Multithread Execution, Callback Group