DTU

Rasmus Andersen

34761 – Robot Autonomy

More on ROS2

Overview of 34761 – Robot Autonomy

- 3 lectures on software frameworks
- 7 lectures on building your own autonomy stack for a mobile robot
- 1 lecture on DL/RL – an overview of black-box approaches to what you have done
- 2 lectures of project work before hand in + guest lectures

Software frameworks			Robot autonomy stack									Project work + guest lectures	
30th Jan	6th Feb	13th Feb	20th Feb	27th Feb	6th Mar	13th Mar	20th Mar	27th Mar	3rd Apr	10th Apr	17th Apr	24th Apr	1st May
Software architecture	Intro to ROS2	More on ROS2	Localization	Mapping	Localization wrt. maps	SLAM	Navigation	Easter	Exploration	Behavior Trees	Black Box Approaches	Group work	

Today

Outline – Recall from last lecture

- The ROS2 framework
  - Nodes
  - Topics
  - Services
  - Actions
- ROS2 tools
- A simple turtle simulation

LAST TIME

- ROS2 workspace
- ROS2 packages
  - Simple publisher and subscriber
- A turtlebot simulation in Gazebo
  - Your environment for the remainder of the course

Recall from last lecture

Topics

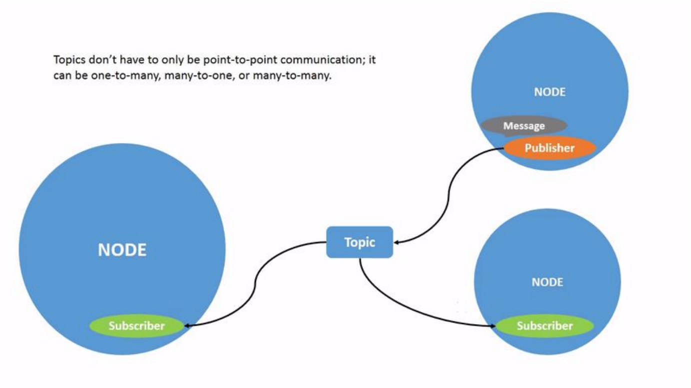
- Continuous data streams
- Data might be published and subscribed at any time independent of any senders/receivers
- Many to many connection

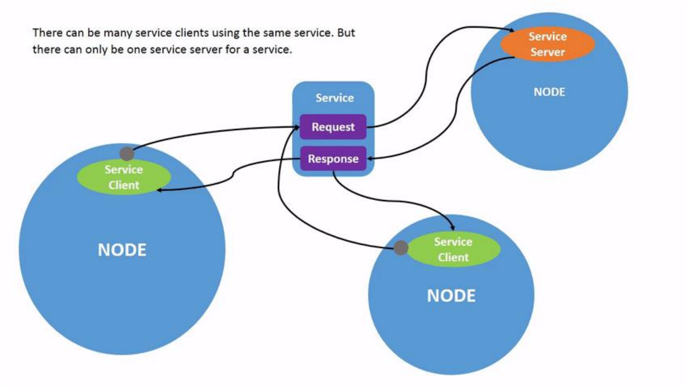
Services

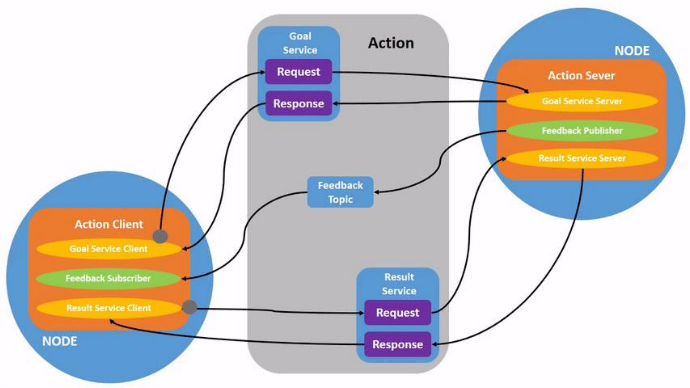
- Semantically similar to function calls
- **<u>Cannot</u>** exit preemptively
- Useful for trigger behavior

Actions

- Provides feedback during execution
- **<u>Can</u>** exit preemptively
- Useful for routines that takes a long time to finish



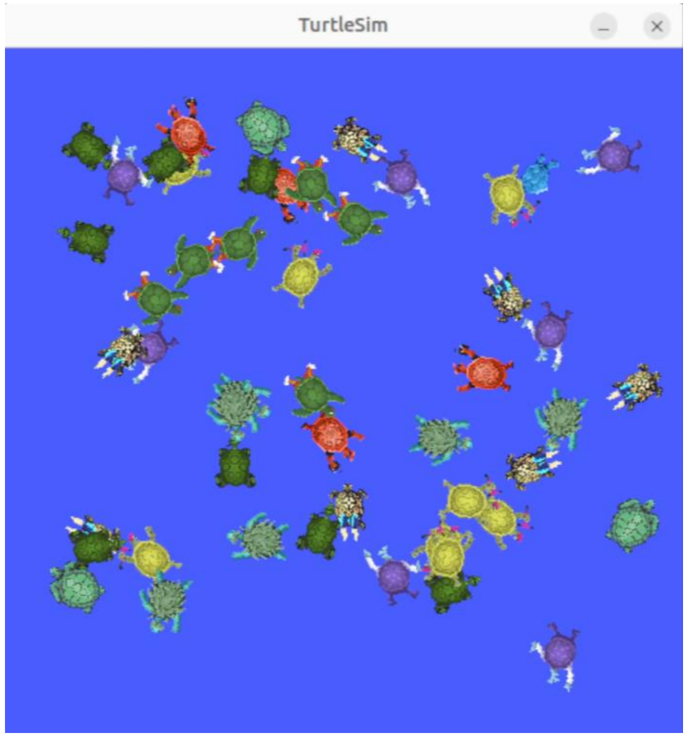




Recall from last lecture

- **Turtlesim Exercise**

1. Kill the turtle already in the simulator
2. Spawn a new turtle at position (6,3), remember to give it a unique name
3. Control the turtle you spawned with the keyboards
4. Record a rosbag of you moving the turtle around
5. Replay the rosbag and see the turtle replicate the movement you recorded
6. Use an action message to control the turtle



Outline

- The ROS2 framework
  - Nodes
  - Topics
  - Services
  - Actions
- ROS2 tools
- A simple turtle simulation
- ROS2 workspace
- ROS2 packages
  - Simple publisher and subscriber
- A turtlebot simulation in Gazebo
  - Your environment for the remainder of the course

TODAY

What is a ROS2 workspace

- A place to store your entire project (source code, build files, resources, etc.)
- A workspace consists of a **build**, **install**, **log**, and **src** folder
  - **The build directory**
    - Intermediate/temporary build files
  - **The install directory**
    - Where the final binaries, libraries, resources, etc. are stored
    - Never modify anything in this directory – it is intended to only be modified by the build system
  - **The log directory**
    - Default location for ROS2 logs
  - **The src directory**
    - Where we put everything to create nodes
    - The only directory we should modify anything in

```
ros2_ws
├── build
├── install
│   └── setup.bash
├── log
└── src
```

Creating a ROS2 workspace

- Creating a ROS2 workspace is simple
  1. Create a directory with the name of your workspace (e.g. ***mkdir ros2_ws***)
  2. Create a directory in your workspace called ***src***
  3. Run ***colcon build*** from your workspace directory

  – The build, install and log directories will automatically be generated

```
ros2_ws
├── build
├── install
│   └── setup.bash
├── log
└── src
```

Using a ROS2 workspace

- Once you have a workspace, source it to get access to your package binaries with ROS2 commands: (***source install/setup.bash***)
  - You must do this every time you add new binaries, libraries, resources, etc. to one of your packages (after building the package)
  - Must be done in every new terminal
  - WHEN IN DOUBT, SOURCE YOUR WORKSPACE
- But why could we run ROS2 packages last time if we didn't have a workspace to source?
  - We used the default setup.bash that gives access to system binaries, libraries, resources, etc.
    - ***source /opt/ros/humble/setup.bash***
  - This command is added to your .bashrc and is run every time you open a terminal (run ***pluma .bashrc*** in a new terminal and scroll to the bottom to see it)

```
ros2_ws
├── build
├── install
│   └── setup.bash
├── log
└── src
```

What is a ROS2 package

- Organizational unit for your ROS2 code
- Package creation in ROS2 uses ament as its build system and colcon as its build tool
- You can create a package using either CMake (for C++) or Python
  - In this course we will primarily use Python
  - You are free to try out creating your packages in C++

```
ros2_ws
├── build
├── install
│   └── setup.bash
├── log
└── src
    ├── my_turtlebot
    │   ├── README.md
    │   ├── launch
    │   ├── maps
    │   ├── models
    │   ├── my_turtlebot
    │   ├── package.xml
    │   ├── resource
    │   ├── rviz
    │   ├── setup.cfg
    │   ├── setup.py
    │   ├── test
    │   ├── urdf
    │   └── worlds
    ├── my_turtlesim
    │   ├── LICENSE
    │   ├── my_turtlesim
    │   ├── package.xml
    │   ├── resource
    │   ├── setup.cfg
    │   ├── setup.py
    │   └── test
    └── robots_are_awesome
        ├── package.xml
        ├── resource
        ├── robots_are_awesome
        ├── setup.cfg
        ├── setup.py
        └── test
```

Creating a ROS2 package

- To create a package, use the convenience tool:
- For python packages:
  - *ros2 pkg create --build-type ament_python <package_name>*
- For Cmake packages
  - *ros2 pkg create --build-type ament_cmake <package_name>*

- *Don’t use “-” in your package or file names. Use “_” as separator instead*

```
ros2_ws
├── build
├── install
│   └── setup.bash
├── log
└── src
    ├── my_turtlebot
    │   ├── README.md
    │   ├── launch
    │   ├── maps
    │   ├── models
    │   ├── my_turtlebot
    │   ├── package.xml
    │   ├── resource
    │   ├── rviz
    │   ├── setup.cfg
    │   ├── setup.py
    │   ├── test
    │   ├── urdf
    │   └── worlds
    ├── my_turtlesim
    │   ├── LICENSE
    │   ├── my_turtlesim
    │   ├── package.xml
    │   ├── resource
    │   ├── setup.cfg
    │   ├── setup.py
    │   └── test
    └── robots_are_awesome
        ├── package.xml
        ├── resource
        ├── robots_are_awesome
        ├── setup.cfg
        ├── setup.py
        └── test
```

```
ubuntu@3d5e2efc84e2:~/ros2_ws/src$ ros2 pkg create --build-type ament_python robots_are_awesome
going to create a new package
package name: robots_are_awesome
destination directory: /home/ubuntu/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['ubuntu <ubuntu@todo.todo>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
creating folder ./robots_are_awesome
creating ./robots_are_awesome/package.xml
creating source folder
creating folder ./robots_are_awesome/robots_are_awesome
creating ./robots_are_awesome/setup.py
creating ./robots_are_awesome/setup.cfg
creating folder ./robots_are_awesome/resource
creating ./robots_are_awesome/resource/robots_are_awesome
creating ./robots_are_awesome/robots_are_awesome/__init__.py
creating folder ./robots_are_awesome/test
creating ./robots_are_awesome/test/test_copyright.py
creating ./robots_are_awesome/test/test_flake8.py
creating ./robots_are_awesome/test/test_pep257.py
```

Creating a ROS2 package

- We now have an empty ros2 package with a python package nested in it – lets create our first node

1. Create a new file in the python package directory
2. Fill in your python code (remember to save)
3. Add your python executable to the build system in the setup.py
4. Build it: ***colcon build***
5. Source the workspace: ***source install/setup.bash***
6. Run the node from your package with
   – ***ros2 run <package name> <name of executable>***

Let's try it out!

Disclaimer: Live demos may contain traces of unexpected errors and sudden disappearances of functionality.

```
ros2_ws
├── build
├── install
│   └── setup.bash
├── log
└── src
    ├── my_turtlebot
    │   ├── README.md
    │   ├── launch
    │   ├── maps
    │   ├── models
    │   ├── my_turtlebot
    │   ├── package.xml
    │   ├── resource
    │   ├── rviz
    │   ├── setup.cfg
    │   ├── setup.py
    │   ├── test
    │   ├── urdf
    │   └── worlds
    ├── my_turtlesim
    │   ├── LICENSE
    │   ├── my_turtlesim
    │   ├── package.xml
    │   ├── resource
    │   ├── setup.cfg
    │   ├── setup.py
    │   └── test
    └── robots_are_awesome
        ├── package.xml
        ├── resource
        ├── robots_are_awesome
        ├── setup.cfg
        ├── setup.py
        └── test
```

Quick summary of what we have learned so far

- We now know how to..

  .. create a workspace

  .. create a package

  .. create an executable

  .. build the ROS2 workspace and source it

  .. how to run an executable with ROS2 commands

Structuring a ROS node

- Generally simple OOP approaches
- This is best taught through the online tutorial (homework for today)

Which one is the publisher?

```python
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1


def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()


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

```python
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)


def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


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

The same goes for services

```python
from example_interfaces.srv import AddTwoInts

import rclpy
from rclpy.node import Node


class MinimalService(Node):

    def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(AddTwoInts, 'add_two_ints', self.add_two_ints_callback)

    def add_two_ints_callback(self, request, response):
        response.sum = request.a + request.b
        self.get_logger().info('Incoming request\na: %d b: %d' % (request.a, request.b))

        return response


def main():
    rclpy.init()

    minimal_service = MinimalService()

    rclpy.spin(minimal_service)

    rclpy.shutdown()


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

Which one is the server?

```python
import sys

from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node


class MinimalClientAsync(Node):

    def __init__(self):
        super().__init__('minimal_client_async')
        self.cli = self.create_client(AddTwoInts, 'add_two_ints')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')
        self.req = AddTwoInts.Request()

    def send_request(self, a, b):
        self.req.a = a
        self.req.b = b
        self.future = self.cli.call_async(self.req)
        rclpy.spin_until_future_complete(self, self.future)
        return self.future.result()


def main():
    rclpy.init()

    minimal_client = MinimalClientAsync()
    response = minimal_client.send_request(int(sys.argv[1]), int(sys.argv[2]))
    minimal_client.get_logger().info(
        'Result of add_two_ints: for %d + %d = %d' %
        (int(sys.argv[1]), int(sys.argv[2]), response.sum))

    minimal_client.destroy_node()
    rclpy.shutdown()


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

Project simulation

- Available at: https://github.com/RasmusAndersen/RobotAutonomy.git

- Run it by doing
  1. *export ROS_DOMAIN_ID=11*
  2. *export TURTLEBOT3_MODEL=burger*
  3. *export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:$(ros2 pkg prefix my_turtlebot)/share/my_turtlebot/models*
  4. *export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:$(ros2 pkg prefix turtlebot3_gazebo)/share/turtlebot3_gazebo/models*
  5. *ros2 launch my_turtlebot turtlebot_simulation.launch.py*
- You can find more information here:
  – https://ros2-industrial-workshop.readthedocs.io/en/latest/_source/navigation/ROS2-Turtlebot.html
- If it doesn’t work at first, you are probably just missing dependencies. Install with:
  – *sudo apt install ros-humble-turtlebot3**

Exercises

- Turtlesim – Create a ROS node that:
  1. Spawns 10 turtles using the *spawn* turtle service from last week
  2. Make all the turtles move in circles using the *cmd_vel* topic
- Turtlebot – install the simulation we use for the rest of the course:
  1. Clone the github repository containing the turtlebot simulation
     git clone https://github.com/RasmusAndersen/RobotAutonomy.git
  2. Build it with colcon
  3. Launch the simulation
  4. Visualize the LIDAR sensor data and TF frames in rviz
  5. Create a ROS node that subscribes to the LIDAR data
- Self-study: learn about transforms by following the tutorial:
  – https://ros2-industrial-workshop.readthedocs.io/en/latest/_source/navigation/ROS2-TF2.html