Webots with ROS2 Custom Controller Documentation

V1.0

By Noah Grzywacz

May 18, 2021

Table of Contents:

1. Requirements Overview and Reference Material:

Before beginning this project, the reader should be familiar with a few things so that they can better grasp the conceptual side of the ROS and Webots relationship as well as some technical skills that will make creation and development of projects much easier.

Conceptual Skills/Knowledge:

- Readers should have at least a basic understanding of the producer/consumer or client/server paradigm that is found in many computer systems and applications, including ROS. This basic paradigm is broken down into two entities: the producer and the consumer. Both entities communicate with one another and pass data between them; however, the producer is the main source of this data in which it performs logic and computations to send data to the consumer which uses that data for some purpose. This paradigm most notably is known in web applications where you have the client, or user, and a server, website/application/etc. The data that travels between the two, in this example, is a request coming from the user to obtain the URL of whatever website they are trying to access. The server side sees this request and, depending on the permissions of the user, either returns the URL so that the user can access the site or returns some sort of error. We will explain more how this paradigm fits into the ROS/Webots relationship later.
- The reader should also have an understanding about controlling robots. In this documentation, we are dealing with a non-holonomic robot which is a robot whose controllable degrees of freedom is less than the total degrees of freedom. The robot implemented in this documentation, the e-puck, has two degrees of freedom: acceleration/braking and turning. The reader should understand that these are the only ways in which we can manipulate the behavior of this kind of robot. That is, we can only control the robot using speeds at which we set the wheels.

Technical Skills:

- The reader should have some experience using object-oriented programming or at the least understand it. Many facets of ROS use objects including communication, execution, computations, and others. The reader should have a solid grasp of OOP so that they can both understand the workings of ROS as well as implement their projects using its technologies.
- The reader should also have a basic understanding of command line scripting and comfort with some sort of text editor. There is no complex bash scripting involved within the project, but we will be using it a lot for installation, building, and running our projects. The editor used for this documentation is VS Code but any text editor should work fine.

Although these skills are not required, they are highly recommended to have at least fundamental knowledge with these skills so that the reader is in a good position to understand the implementation of the controller outlined in this documentation and go on to create their own.

Lastly, there will be many references throughout this documentation to other sources where content and knowledge is pulled from. There is no single resource in which this documentation is based off of, however, there are some predominant ones that were used in the making of both this documentation and the project implemented within it. These two main resources are:

- Soft-Illusion's Webots ROS2 Tutorial Series: This is a phenomenal playlist of videos that explains everything from installing ROS2 with Webots to implementing some of the most complicated technologies included in the ROS package. Much of the section Downloading, Installing, and Building ROS2 with Webots uses the first two videos of the series as reference, with a few modifications.
- The second large resource used in the implementation of the project outlined in this documentation is the ROS documentation itself, found here. Note that this is the documentation for the original ROS installation, not ROS2. Much of the information contained in these docs should also pertain to ROS2 with a few changes.

If the reader is ever at a loss of understanding or would like more information about the current topic, these two sources should be the first things the reader looks to. There will be additional resources used throughout the documentation that will be referenced as they come up, but these are the main two. With all that said and done, let's get to it!

2. Creating A Package:

This section of the documentation is dedicated to teaching you how ROS2 packages work and how to create them. We will start with a brief explanation of ROS2 packages, the role they play, and how useful they are. Then we will check out ROS's tutorial for creating packages quickly and simply. Lastly, we look at creating a package for a specific task with ROS2 and Webots.

2.1 A Brief Introduction to ROS Packages:

Packages are the basic building blocks for every piece of software in ROS. From datasets and configuration files to third-party software, almost every logical and useful piece of software in ROS is part of a package module. This has 2 main advantages, among others. The first and biggest advantage of a package structure is modularity. Having modular pieces of a bigger system makes every step of the development process easier. From abstracting functionality to implementing and debugging, modular components allow for easier segmentation and individual development which can result in a more fluid development process and improve efficiency. The second big advantage of the ROS package structure is portability. We do not directly employ this advantage of ROS in this documentation, however because each package is individual, we can move packages between systems very easily and run them if the system already have the base installation of ROS. This becomes extremely useful when you have multi-system robotics projects where you may have the logic and calculations of for robot odometry on one system and the actual controller on another. These packages come with other advantages such as ease-of-use and reusability which also aid in the development and implementation process.

2.2 Creating your first ROS2 Package:

To begin understanding more about packages, we'll first go into a simple introduction on creating our own packages. The instructions below follow the ROS2 documentation for "Creating your first ROS2 package" found here. If you get lost or encounter an error that this documentation does not address, or even would just rather follow their tutorial, feel free to check it out using the link provided.

Procedure:

1.) Before creating our package, we need have a workspace to work in. Luckily, you can use the `ros2_ws` workspace you created in the previous chapter. Open a Terminal window and move into the source directory of your workspace using:

```
cd ~/ros2_ws/src
```

2.) From here, you can create your new package using the following command:

```
ros2 pkg create --build-type ament_python --node-name my_node my_package
```

This command will create a simple Hello World package because of the `--node-name` argument. The general ROS2 package creation syntax is as follows:

```
ros2 pkg create --build-type ament_python <package_name>
```

When you press ENTER after entering the first command, you should see some messages showing the automatic generation of your package files. If you type `ls` you should also see a new directory named `my_package` in your `src` folder. If an error occurs in this step, you may need to re-source your ROS installation or append some additional environment variables to your .bashrc script. Refer to the ROS documentation using the link at the beginning of this section if this occurs. It is not covered in this documentation because we have already sourced our installation based on the special commands we already appended to our .bashrc script.

3.) Next, you will need to build the package. Whenever you build your packages, you will want to be in the base directory of your workspace, so move back to the base directory using the following command:

```
cd ~/ros2_ws/
```

Then run the command:

```
colcon build
```

Whenever you modify files in your package and want to test those modifications you will need to rebuild the package. If you have a lot of packages in your workspace, it can take a long time for your computer to finish running a single `colcon build` command. For this reason, it's often useful to specifically select which packages you want to build. To do this you can simply add the optional `--packages-select` argument to the previous command like so:

```
colcon build –packages-select my_package
```

This is a command that you will be using a lot when developing and testing your own packages so remember it well.

4.) Afterwards, open a new terminal and move into your `ros2_ws` workspace using the same `cd` command above. You will need to source your workspace using the following command:

```
. install/setup.bash
```

This command simply runs the setup script for your workspace that is present in the install folder of your workspace. Whenever you rebuild your package, you will need to source it again so this is another command that you should remember and become familiar with.

5.) Now that you have your package set up and sourced, you can run it using the following command:

```
ros2 run my_package my_node
```

Which should print the message:

```
Hi from my_package.
```

Before we move on to implementing a project package with a specific goal in mind, we should take a look at the files and file structure of packages in ROS to get a better understanding the role that each file plays in running an ROS2 project with Webots.

6.) Since you should still be in the base directory of your workspace, move into the package directory you just created using the command:

```
cd /src/my_package/
```

Now type the `ls` command to list all the files and directories in your present directory. You should see:

```
my_package  package.xml  resource  setup.cfg  setup.py  test
```

Each of these files have a specific purpose. We will specifically look at the `package.xml` and `setup.py` files. Before doing that, we should mention the `my_package/` sub-directory within your package directory. The directory contains your `my_node.py` file. The concept of nodes is something that we will expand on further in the next section when we create a project package with a specific goal in mind. For now, just know that any nodes you create in the future will go in this sub-directory. Now we will look at the `package.xml` file. To open this file in VS Code, simply type the command:

```
code package.xml
```

When you open VS Code, you should see something like the following:

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>my_package</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="<Your Username>@todo.todo">Your Username</maintainer>
  <license>TODO: License declaration</license>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

This file contains information about the package including the name, description, and maintainer, It also includes release information. However, most importantly, this file also contains all the references to the dependencies required to run the package.

If you plan on releasing any of the packages that you create, you need to declare the license associated with it, and it would also be helpful to include a summarized description to give users an idea of what your package does.

7.) Next, we will take a look at the `setup.py` file. Start by closing VS Code and in the same Terminal window you were in before, run the command:

```
code setup.py
```

When you open that file, you should see something like the following:

```
from setuptools import setup

package_name = 'my_package'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='<Your Username>',
    maintainer_email='<Your Username>@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'my_node = my_package.my_node:main'
        ],
    },
)
```

This file contains similar information about the name, maintainer, and description. This file is a little bit more involved with elements such as required packages, data files, and entry points for our package. We will be adding more modifications to it in the next section to have all the necessary elements to setup, build, and run our project package.

This is the end of this section. Hopefully, you have gotten at least a basic understanding of the role that packages play in ROS2 as well as how to create them and some of the roles that the files in the packages play. In the next section, we will present a certain task and then create a package to accomplish the task. In the following chapter, we will go into depth into how to implement your robot controller using ROS2 nodes to communicate. However, first we need to create the package that this controller will be present in.

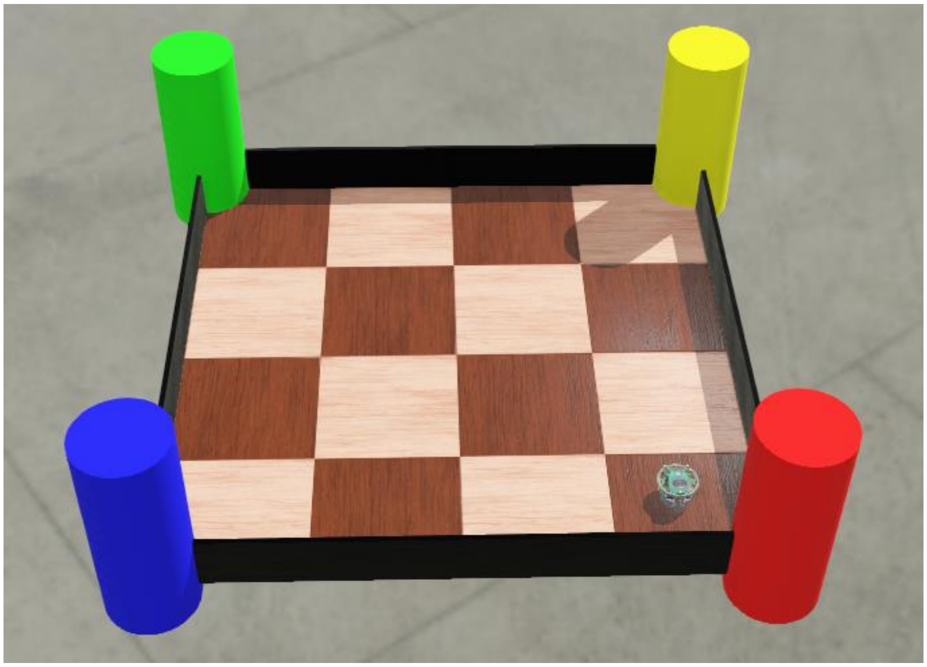
2.3 Creating a Project Package Given a Certain Task:

This section, and following chapter covers the original intent and purpose for this documentation. In this section, we will be given a task that a robot needs to complete. We will begin setting up the custom package in this section as well as going over each part of it in detail to describe how the package supports the robot controller and interfaces with Webots through ROS2. If you wish to follow along in the creation of this package, you may. However, the resources and files for this package should already be present in the folder that this documentation came in or the github repository of this project. Because of this, this section and the next chapter will not follow the previous scheme of providing tutorials and procedures. Rather, it will be closer to typical documentation that describes the flow of logic and how the program operates.

The Task:

Given a robot in world composed of 16 tiles and 4 uniquely colored cylinders, we need to create a package that allows the robot to localize based on its position relative to the cylinders as well as navigate through each of the 16 tiles until it has visited every tile. You may have already seen this task in a robotics course if you took one. However, there is an additional requirement for this project: it must be implemented using ROS2 where we break up the controller logic and world information gathering into two separate modules. We will discuss more on this breaking-up of roles in the next chapter when we implement our controller.

The world described can be seen below:



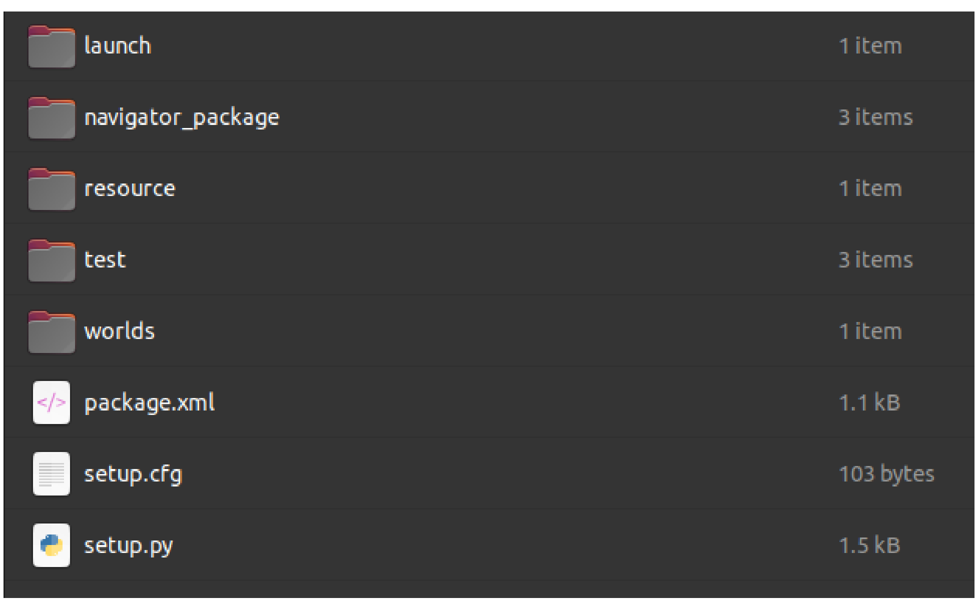
This task addresses many of the basic functions and problems found in robotics, including localization, tracking, odometry, and use of various sensors. To complete this task, we will be using an assortment of different sensors that you may or may not already be experienced with. These sensors are listed below along with a brief description of what they detect.

Sensors:

- Distance Sensors: We have 3 distance sensors on our E-puck robot: one in front, one on the right side, and one on the left side. These infrared sensors send out a beam of light in their respective directions to detect surfaces and objects adjacent to the robot. In Webots, these sensors can have an assigned limited range as well as a noise associated with them. However, for the purposes of this task, we will assume the range is unlimited and there is no noise on the sensor. The units for this sensor are returned in meters.
- Position Sensors: We have 2 position sensors on our E-puck robot: one for the left wheel, and one for the right wheel. The position sensor tracks the distance that each wheel has travelled in radians. Like the distance sensor, noise can be added to this component to simulate more realistic robots, but we will not have any assigned for this task. We do not use this sensor much in this task, however, it is good to know all the possible components we can use to our advantage.
- Inertial Measurement Unit: The Inertial Measurement Unit, or IMU, of the E-puck is used to measure angles of rotation on each of the axes in the three-dimensional coordinate system. Because we are dealing with a two-dimensional robot (can only move in x and y), we are only concerned with the angle about the z-axis called the “yaw.” The IMU is an important sensor for helping the robot localize and maintain an accurate calculation of its relative position when moving.
- Camera: The last and most significant sensor used in this task is the camera. The camera sensor generates an image of the whatever part of the world that the robot is facing. The camera can detect specific objects called Recognition Objects that helps us a lot in allowing the robot to localize.

Package Creation and Setup:

The project package for this task was named `navigator_package` and created in the same manner as described in the previous section. Within this package we have many of the same base files and directories you find in almost every ROS2 Webots package, as shown below:



This package has two additional sub-directories that we did not see in the previous section: the `launch` directory and the `worlds` directory. Both directories are common to find in other more complex ROS2 packages. The `launch` directory contains the python script to launch our project package in Webots which we will look at shortly. The `worlds` directory contains, as the name implies, the world files for world associated with this task.

In this section, we will look at the three most significant files that make up our project package: the `navigator_launch.py` file, the `package.xml` file, and the `setup.py` file. We will start with the `package.xml` file.

`package.xml`

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>navigator_package</name>
  <version>1.0.0</version>
  <description>Package to run Webots tasks with ROS2</description>
  <maintainer email="noah-ai@todo.todo">noah-ai</maintainer>
  <license>Apache License 2.0</license>

  <exec_depend>rclpy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
  <exec_depend>nav_msgs</exec_depend>
  <exec_depend>sensor_msgs</exec_depend>
  <exec_depend>geometry_msgs</exec_depend>
  <exec_depend>webots_ros2_msgs</exec_depend>
  <exec_depend>builtin_interfaces</exec_depend>
  <exec_depend>tf2_ros</exec_depend>
  <exec_depend>webots_ros2_core</exec_depend>
  <exec_depend>rviz2</exec_depend>


  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>

```

The contents of this package are like that of the basic `package.xml` file we saw in the previous section. However, notice that a few additional dependencies have been appended. Most notably, we include `webots_ros2_core`, the base package for all things ROS2 and Webots related, `std_msgs` and `sensor_msgs,` to pass the information that we receive from our sensors to our controller logic file, as well as `webots_ros2_msgs,` which provides data types for specific elements in Webots that we can send to different parts of our robot controller.

The `rclpy` dependency is another important inclusion in our package. In order to break up the functionality of our code but still have the two modules communicate, we need “spin” the separate nodes which is a class function of `rclpy.`

setup.py

```python
from glob import glob
from setuptools import setup

package_name = 'navigator_package'

data_files = []
data_files.append(('share/ament_index/resource_index/packages', [
    'resource/' + package_name
]))
data_files.append(('share/' + package_name + '/worlds', [
    'worlds/lab3_task2.wbt'
]))
data_files.append(
    ('share/' + package_name + '/protos/icons', glob('protos/icons/*')))
data_files.append(
    ('share/' + package_name + '/worlds/textures', glob('worlds/textures/*')))
data_files.append(
    ('share/' + package_name + '/protos/textures', glob('protos/textures/*')))
data_files.append(('share/' + package_name, [
    'package.xml'
]))
data_files.append(('share/ament_index/resource_index/packages',
    ['resource/' + package_name
]))

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=data_files,
    install_requires=['setuptools', 'launch'],
    zip_safe=True,
    maintainer='noah-ai',
    maintainer_email='noah-ai@todo.todo',
    description='Package for running Webots tasks with ROS2',
    license='Apache License 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'enable_robot = navigator_package.slave:main',
            'navigator = navigator_package.master:main',
        ],
        'launch.frontend.launch_extension': ['launch_ros = launch_ros']
    },
)
```

As we stated earlier, `setup.py` is more involved than the `package.xml` file. Because we are dealing with a larger, and more complex package, we need to include all the necessary data files for the world that we want to simulate, as well as the additional files we have created to help run our package. The `setup.py` file is run whenever we build our package so updated versions of all the data files and world files are added to the base `install` folder of our workspace which is where the package is run from. These essential files are added to the initially empty list `data_files` to be used in the `setup` function below.

Another large part of the setup file and function is the entry points at the bottom. These entry points are what we use to “spin up” the separate nodes of our controller structure called `master` and `slave`. We will see how these entry points are activated in the `navigator_launch.py` launch file below.

`navigator_launch.py`

```python
import os
from launch.substitutions.path_join_substitution import PathJoinSubstitution
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch import LaunchDescription
from ament_index_python.packages import get_package_share_directory
from launch_ros.actions import Node


def generate_launch_description():
    package_dir = get_package_share_directory('navigator_package')
    core_dir = get_package_share_directory('webots_ros2_core')
    webots = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(core_dir, 'launch', 'robot_launch.py')
        ),
        launch_arguments=[
            ('package', 'navigator_package'),
            ('executable', 'enable_robot'),
            ('world', PathJoinSubstitution(
                [package_dir, 'worlds', 'lab3_task2.wbt'])),
        ]
    )

    Nav = Node(
        package='navigator_package',
        executable='navigator',
        name='master_node'
    )

    return LaunchDescription([
        webots,
        Nav
    ])
```

The `navigator_launch.py` file contains the basic `generate_launch_description` to launch our package. We provide it first with the name of the package and the core directory, which is always `webots_ros2_core`. We then create the launch description for Webots that calls the base Webots launch file to launch Webots with the launch arguments including the package, the entry point, and the world file to use. The `executable enable_robot` is the entry point that we saw in our `setup.py` file that activates the `slave` of our controller. We also create a launch node called `Nav` in which we again provide the package name for reference, the entry point `navigator,` and the name of the node. We then launch these two components to run Webots with our two node modules.

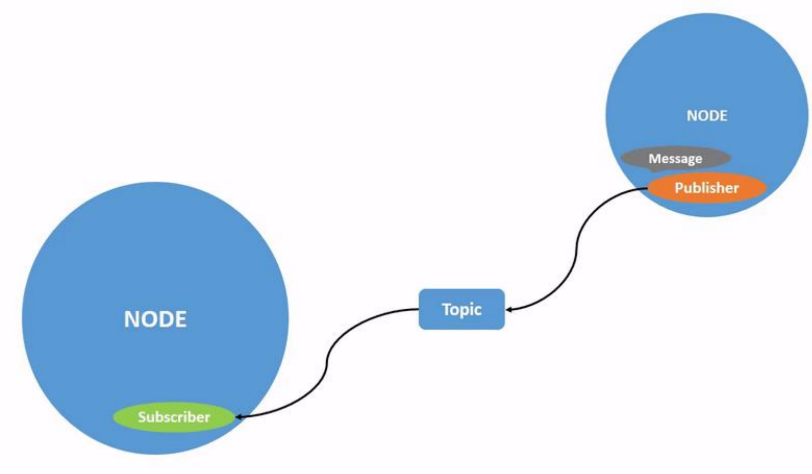
This is the end of this section. If need more information on the other pieces of the package or just want to take a closer look at each of the files, be sure to check out the github repo for this project. In the next chapter we will see how to implement the controller to accomplish this task into our `master` and `slave` files.

3. Implementing the Controller:

This chapter is a continuation of the last section of the last chapter in which we implement the controller for our robot to accomplish the task assigned. In this chapter, we will specifically look at the modules that outline our master and slave nodes. These files are in the `/navigator_package` subdirectory in our base package directory.

Before we begin looking at these files, it would be beneficial to discuss more about the interaction of these two nodes and how they transfer information. As was stated at the beginning of this documentation, the controller for this project follows a producer-consumer or master-slave paradigm where information is transferred between modules to divide up the work that each module does. This, like the package scheme of ROS, helps us organize our projects better and create more modular components that we can then reuse on similar projects.

For this specific project, the `slave` node that enabled with the launch entry point `enable_robot` is launched with Webots. Thus, this is the node that will be gathering all the sensors and world information from Webots to pass to the `master` node which contains the logic to complete the assigned task. The way that these nodes communicate is through the ROS abstracted publishers and subscribers. Somewhat similar to the example that we did in chapter 4 when we tested our ROS installation by launching a `talker` and `listener`, the `slave` node is analogous to the `talker` and the `master` node is analogous to the `listener`. However, instead of feeding information one way as we did with the `talker` and `listener,` we need to have a bidirectional transfer of information between `master` and `slave` so that we can transfer the sensor and world information from Webots to the `master` node through the `slave` node so that it can perform its controller logic and calculations, and send the proper commands back to Webots through the `slave` node to see the robot simulate and complete the task. A diagram showing the basic architecture of this relationship is shown below:



Source: https://docs.ros.org/en/foxy/Tutorials/Topics/Understanding-ROS2-Topics.html

We will start by looking at `slave.py.` We will not be going through the file line by line because a lot of the sections use the same scheme, but we will be covering the overall structure and key elements that go into the file and its communication. You may refer to the direct file and all of its contents by opening it in the github repository or text editor.

slave.py

The `slave.py` file has three main parts:

1.) The instantiation of Webots sensors and publishers

2.) Sensor callbacks and publishing of sensor values

3.) Assigning received wheels velocities from `master` to Webots

Starting with the first part, each of the sensors that we use in our package must be instantiated within the package. We cannot start reading data from the Webots world when there is no medium to read it through. Each of the sensors in the `slave` file is essentially assigned as a Webots sensor object using the `.getDevice()` method.

We also need to enable this Webots sensor object using the `.enable()` with the timestep of the simulation as the argument. The timestep of a simulation represents the time, in milliseconds, that the simulation spends in one step. Each step of a simulation can be described as the computations or logic for every simulated object. So, if we have a timestep of 32 for a robot controller that just tells the robot to move forward, the simulation will tell the robot to move forward every 32 milliseconds. This concept is like frames per second.

The last thing we need to do for each of these sensors is create publisher to publish the values they receive to the `master` node. The function `create_publisher()` takes in the type of data that needs to be transmitted, which will be different for the different sensors, and the name of the publisher that the subscriber in the `master` node will be subscribing to.

This same scheme is followed for every sensor in the first part of `slave` file. A sample of this scheme is shown below with the IMU as the sensor:

```
        self.imu = self.robot.getDevice('inertial unit')
        self.imu.enable(self.timestep)
        self.imu_publisher = self.create_publisher(Float64, 'imu_SENS', 1)
```

The second main part of the `slave` node is the sensors callbacks which retrieve the sensor readings from Webots to store in publishable message variables. These messages receive their type from the what we declared in our instantiation of the message publisher. We get the data from Webots using the sensor object function `.getValue()` to assign this message variable with the sensor data. Once we have the data, we publish it to the `master` node using the publisher we created. This scheme for all, except the camera, sensor callbacks is shown below with the front distance sensor used as the sensor example:

```
        msg_distance_front = Float64()
        msg_distance_front.data = self.front_distance_sensor.getValue()
        self.front_distance_sensor_publisher.publish(msg_distance_front)
```

The majority of the sensor_callback() member function is shown in the figure below, the remaining part of the function is shown in the next figure following it:

```python
# Sensor callback that declares sensors messages, populates with data, and publishes to topics
def sensor_callback(self):
    # Publish distance sensor value
    msg_distance_right = Float64()
    msg_distance_right.data = self.right_distance_sensor.getValue()
    self.right_distance_sensor_publisher.publish(msg_distance_right)

    msg_distance_front = Float64()
    msg_distance_front.data = self.front_distance_sensor.getValue()
    self.front_distance_sensor_publisher.publish(msg_distance_front)

    msg_distance_left = Float64()
    msg_distance_left.data = self.left_distance_sensor.getValue()
    self.left_distance_sensor_publisher.publish(msg_distance_left)

    #Publish position sensor value
    msg_position_right = Float64()
    msg_position_right.data = self.right_position_sensor.getValue()
    self.right_position_sensor_publisher.publish(msg_position_right)

    msg_position_left = Float64()
    msg_position_left.data = self.left_position_sensor.getValue()
    self.left_position_sensor_publisher.publish(msg_position_left)

    #Publish imu yaw sensor value
    msg_imu_yaw = Float64()
    msg_imu_yaw.data = self.imu.getRollPitchYaw()[2]
    self.imu_publisher.publish(msg_imu_yaw)
```

As was stated above, the camera requires additional logic to get the information about the Recognition Objects that we need to initially localize the robot. Although we can detect Recognition Objects with our instantiated camera device, we cannot directly send a Recognition Object to the `master` node for the information to be extracted and used. Because of this, we need to do some pre-processing in the `slave` file to extract the information we need from the Recognition Objects then send that data to the `master` file using a supported publishing type. The logic for this is shown below and will be explained in more depth afterward.

```python
        ids = None
        pos = None
        colors = None
        msg_camera_image = Image()
        msg_camera_data = Float64MultiArray()

        msg_camera_image.data = self.camera.getImage()
        if self.camera.getRecognitionObjects():
            colors = self.camera.getRecognitionObjects()[0].get_colors()
            pos = self.camera.getRecognitionObjects()[0].get_position()
            ids = self.camera.getRecognitionObjects()[0].get_id()
            msg_camera_data.data = [float(ids), pos[0], pos[1], pos[2], colors[0], colors[1], colors[2]]
        self.camera_image_publisher.publish(msg_camera_image)
        self.camera_data_publisher.publish(msg_camera_data)
```

We start by initializing three variables with `None` as well as declaring two different camera messages with types `Image` and `Float64Array`, respectively. The camera message `msg_camera_image` is just assigned the image that the camera is capturing. Then we test if the camera is detecting any Recognition Objects and if it is we assign each of previously defined variables with the object color, position, and id. We then parse that information into the `Float64Array` data array message and publish both camera messages. Because we publish two different camera messages with different types, we have to create two different publishers as shown below:

```
        self.camera = self.robot.getDevice('camera1')
        self.camera.enable(self.timestep)
        self.camera.recognitionEnable(self.timestep)
        self.camera_image_publisher = self.create_publisher(Image, 'camera_image_SENS', 1)
        self.camera_data_publisher = self.create_publisher(Float64MultiArray, "camera_data_SENS", 1)
```

Note: We also need to enable detection of Recognition Objects for this sensor.

The last part of the `slave` file is the subscriber and associated callback function that receive the calculated wheel speeds from the `master` node and assign these wheel speeds to the robot in Webots. We start by creating a subscriber to receive the data from `master` as shown below:

```
        self.cmd_vel_subscriber = self.create_subscription(
            Twist, 'cmd_vel', self.cmdVel_callback, 1)
```

This declaration of a subscriber follows the same structure of a publisher declaration where it declares the type of data that the subscriber is receiving, the name of the subscriber, and one additional argument: the callback function used to retrieve the passed information. The type `Twist` shown above is a special message type imported from `geometry_msgs` that is comprised of two 3-dimensional vector objects, a linear and an angular. The callback function associated with this subscriber is where we unpack the data contained in the transferred message and assign the wheel speeds to the robot in Webots.

```
    def cmdVel_callback(self, msg):
        wheel_gap = 0.057912 # in meter
        wheel_radius = 0.02032  # in meter

        left_speed = ((2.0 * msg.linear.x - msg.angular.z *
                       wheel_gap) / (2.0 * wheel_radius))
        right_speed = ((2.0 * msg.linear.x + msg.angular.z *
                        wheel_gap) / (2.0 * wheel_radius))
        left_speed = min(self.motor_max_speed,
                         max(-self.motor_max_speed, left_speed))
        right_speed = min(self.motor_max_speed,
                          max(-self.motor_max_speed, right_speed))

        self.left_motor.setVelocity(left_speed)
        self.right_motor.setVelocity(right_speed)
```

To assign the wheel speeds with the proper values we need to use the vector object values from the transferred message, some of the robot specifications, and the max speed that the robot wheels can move at. After completing these calculations, we simply assign the speeds to the wheels in Webots.

`master.py`

The `master.py` file is where we implement our controller for the robot to complete its task. Like the `slave.py` file we will not be going through the file analyzing what it does line by line. Rather, we will be looking at the overall structure and operation. First, however, we will look at how this file interacts with the `slave.py` file through it's subscribers and callback functions. Similar to how we need the publishers in the `slave.py` file to publish data, we need subscribers in the `master.py` file to receive the data. Each of the publishers created in the `slave.py` file must have a subscriber to receive the data. The subscribers for each of these publishers is shown below:

```
        self.rightDS_sub = self.create_subscription(Float64, 'right_IR', self.rightDS_callback, 1)
        self.leftDS_sub = self.create_subscription(Float64, 'left_IR', self.leftDS_callback, 1)
        self.frontDS_sub = self.create_subscription(Float64, 'front_IR', self.frontDS_callback, 1)
        self.rightPOS_sub = self.create_subscription(Float64, 'right_POS', self.rightPOS_callback, 1)
        self.leftPOS_sub = self.create_subscription(Float64, 'left_POS', self.leftPOS_callback, 1)
        self.imu_sub = self.create_subscription(Float64, 'imu_SENS', self.imu_callback, 1)
        self.camera_image_sub = self.create_subscription(Image, 'camera_image_SENS', self.camera_image_callback, 1)
        self.camera_data_sub = self.create_subscription(Float64MultiArray, 'camera_data_SENS', self.camera_data_callback, 1)
```

Like the subscriber created in the `slave.py` file to assign wheel speeds to the robot in Webots, each of these subscribers are declared with the type of data it will take, the publisher to read from, and the callback function to extract the transferred data. For example, the IMU callback function is shown below, which simply extracts the IMU information from the transferred `msg` variable and assigns it to a local variable.

```
    def imu_callback(self, msg):
        self.yaw = msg.data
```

The other callbacks generally follow the same structure:

```
def rightDS_callback(self, msg):
    self.rds = msg.data
    self.NavigatorModule()

def frontDS_callback(self, msg):
    self.fds = msg.data

def leftDS_callback(self, msg):
    self.lds = msg.data

def rightPOS_callback(self, msg):
    self.rpos = msg.data

def leftPOS_callback(self, msg):
    self.lpos = msg.data

def imu_callback(self, msg):
    self.yaw = msg.data

def camera_image_callback(self, msg):
    self.camera = msg.data

def camera_data_callback(self, msg):
    if msg and msg.id not in self.target_ids:
        #self.target_ids.append(msg.data[0])
        self.target_ids.append(msg.id)
        self.target_colors.append(msg.colors.data.tolist())
        self.targets_positions.append(msg.position.data.tolist())
        #self.targets_positions.append(msg.data[1:4].tolist())
        #self.target_colors.append(msg.data[4:7].tolist())
        self.recognition_objects.append(msg)
```

Once we have all the sensor data, we can perform our controller logic. As of this version of the documentation, the controller operates as a finite state machine with four different states:

1.) Triangulate – Initially assigned state in which the robot spins 360 degrees to locate Recognition Objects and use embedded data to determine its initial pose.

2.) Straight – State in which the robot moves forward in straight line while updating its position as it moves.

3.) Turn – State in which the robot turns 90 degrees be evaluating its current yaw, setting a goal yaw corresponding to whatever direction is more favorable, and turning until it reaches that yaw.

4.) Stop – Final state in which the robot has traversed and tracked all cells in the world. In this state the robot does not move and just prints a final message notifying the user that all cells have been tracked.

The state machine of this file resides in a `NavigatorModule()` within the `Navigator` node which is the `master` node. We call this module within the first subscriber callback as shown below.

```
    def rightDS_callback(self, msg):
        self.rds = msg.data
        self.NavigatorModule()
```

By calling this module in the callback function, we synchronize the execution of the of the `master` node with the `slave` node by aligning the publishing and reception of data.

We will now take a closer look at each of the main function associated with each of these states. Starting with the initial state triangulate which calls the `triangulate()` function.

```python
    #function used to find robot starting position, robot needs to be able to see three cylinders to work
    def triangulate(self):
        self.turnLeft()
        self.get_logger().info("Triangulating position...")

        if self.init == 0 and self.getO() != 0:
            self.gYaw = self.getO()
            self.init = 1

        if len(self.target_ids) < 4:
            self.state = 0
            return

        #triangulation of initial pose calculated here, based on typical triangulation algorithm
        elif len(self.target_ids) >= 4:
            x1, y1 = self.setvars(self.identify(self.target_colors[0]))
            x2, y2 = self.setvars(self.identify(self.target_colors[2]))
            x3, y3 = self.setvars(self.identify(self.target_colors[1]))

            t1 = [i * 39.3701 for i in self.targets_positions[1]]
            t2 = [i * 39.3701 for i in self.targets_positions[3]]
            t3 = [i * 39.3701 for i in self.targets_positions[2]]

            r1 = self.radius(t1)
            r2 = self.radius(t2)
            r3 = self.radius(t3)

            A = (-2 * x1 + 2 * x2)
            B = (-2 * y1 + 2 * y2)
            C = (r1 ** 2) - (r2 ** 2) - (x1 ** 2) + (x2 ** 2) - (y1 ** 2) + (y2 **2)
            D = (-2 * x2 + 2 * x3)
            E = (-2 * y2 + 2 * y3)
            F = (r2 ** 2) - (r3 ** 2) - (x2 ** 2) + (x3 ** 2) - (y2 ** 2) + (y3 **2)

            x = ((C * E - F * B) / (E * A - B * D))
            y = ((C * D - A * F) / (B * D - A * E))

            self.currPose[0] = x
            self.currPose[1] = y
            self.currPose[2] = self.getO()

            #finish turning 360 degrees
            if self.getO() < self.gYaw + 0.01 and self.getO() > self.gYaw - 0.01 and len(self.target_ids) >= 5:
                self.stop()
                self.gYaw = None
                self.state = 1
```

The `triangulate()` function starts by telling the robot to spin to the left to look for Recognition Objects referred to as “targets.” Once it recognizes at least 3 targets, it performs the triangulation calculation to find the initial pose. This calculation is based on a standard triangulation algorithm. A .pdf describing the algorithm can be found in the `docs` folder in the github repo. Once it has calculated this initial pose of the robot, it allows the robot to continue spinning until it completes a full 360 degrees where it changes the state of the FSM to “Straight.”

Once in the “Straight” state, the robot follows the `moveForward()` function shown below.

```
#move forward while checking for obstacles and updating cell memory and current position
def moveForward(self):
    self.goStraight()
    range = self.outer()
    if(self.getDistances()[2] < range):
        self.stop()
        if self.getDistances()[0] > self.getDistances()[1]:
            self.direction = 'LEFT_TURN'
        else:
            self.direction = 'RIGHT_TURN'
        self.state = 2

    self.updatePose()
    self.updateMem()
```

This function is essentially an intermediate wrapper function for the `updatePose()` wrapper and `updateMem()` function. It instructs the robot to move forward while detecting obstacles in front of it. Once the robot gets within a certain range of an obstacle the robot determines the best way to turn based on the space available and changes the state to “Turn.” The `updatePose()` wrapper function calls `updatePos()`, shown below, and `update()` which updates the orientation saved in the current pose. The `updateMem()` function updates the currently tracked cell memory and cell number of the current pose using the current position of the robot and the coordinate bounds of each of the cells.

```
#function to update position of the robot in the world using distance sensor readings
def updatePos(self):

    #update robot position based on mesaurement readings
    if (abs(self.currPose[2]) - PI < 0.1 and abs(self.currPose[2]) - PI > -0.1): #facing up
        self.currPose[1] = 20 - self.getDistances()[2] - ROBOT_RADIUS
        if (self.getDistances()[0] < self.getDistances()[1]):
            self.currPose[0] = self.getDistances()[0] - 20
        else:
            self.currPose[0] = 20 - self.getDistances()[1]
    elif (abs(self.currPose[2]) - (PI / 2) < 0.1 and self.currPose[2] < 0 and
        abs(self.currPose[2]) - (PI / 2) > -0.1 ): #facing left
        self.currPose[0] = -20 + self.getDistances()[2] + ROBOT_RADIUS
        if (self.getDistances()[0] < self.getDistances()[1]):
            self.currPose[1] = self.getDistances()[0] - 20
        else:
            self.currPose[1] = 20 - self.getDistances()[1]
    elif (abs(self.currPose[2]) < 0.1 and self.currPose[2] > -0.1): #facing down
        self.currPose[1] = -20 + self.getDistances()[2] + ROBOT_RADIUS
        if (self.getDistances()[0] < self.getDistances()[1]):
            self.currPose[0] = 20 - self.getDistances()[0]
        else:
            self.currPose[0] = self.getDistances()[1] - 20
    else: #facing right
        self.currPose[0] = 20 - self.getDistances()[2] - ROBOT_RADIUS
        if (self.getDistances()[0] < self.getDistances()[1]):
            self.currPose[1] = 20 - self.getDistances()[0]
        else:
            self.currPose[1] = self.getDistances()[1] - 20
```

The `updatePos()` function uses the current orientation of the robot and distance sensor readings to modify and update the x and y positions of the current pose.

The third and last non-trivial state of the FSM controller is the “Turn” state which is shown below.

```python
    def turn(self, direction):
        #turning left
        if direction == 'LEFT_TURN':
            self.turnLeft()
            #assign goal yaw if not set
            if self.gYaw is None:
                if self.getO() < -1.5 and self.getO() > -1.63: #left
                    self.gYaw = 0
                elif self.getO() < 1.63 and self.getO() > 1.5: #right
                    self.gYaw = PI
                elif (self.getO() < PI + 0.05 and self.getO() > PI - 0.05 or
                    self.getO() > -PI -0.05 and self.getO() < -PI + 0.05): #up
                    self.gYaw = -PI / 2
                elif self.getO() < 0.05 and self.getO() > -0.05: #down
                    self.gYaw = PI / 2
            #just check orientation/yaw to know when to stop turning
            if self.gYaw is not None and self.getO() < self.gYaw + 0.01 and self.getO() > self.gYaw - 0.01:
                self.stop()
                self.gYaw = None
                self.direction = None
                self.state = 1

        #urning right
        elif direction == 'RIGHT_TURN':
            self.turnRight()
            #assign goal yaw if not set
            if self.gYaw is None:
                if self.getO() < -1.5 and self.getO() > -1.63: #left
                    self.gYaw = PI
                elif self.getO() < 1.63 and self.getO() > 1.5: #right
                    self.gYaw = 0
                elif (self.getO() < PI + 0.05 and self.getO() > PI - 0.05 or
                    self.getO() > -PI -0.05 and self.getO() < -PI + 0.05): #up
                    self.gYaw = PI / 2
                elif self.getO() < 0.05 and self.getO() > -0.05: #down
                    self.gYaw = -PI / 2
            #just check orientation/yaw to know when to stop turning
            if self.gYaw is not None and self.getO() < self.gYaw + 0.01 and self.getO() > self.gYaw - 0.01:
                self.stop()
                self.gYaw = None
                self.direction = None
                self.state = 1
```

The `turn()` function takes in a `direction` variable which indicates which direction, right or left, the robot wants to turn. This variable is assigned in the “Straight” state when the robot enters a certain range of an obstacle. It then sets the goal yaw that it wants to turn to based on the current orientation and input `direction` variable. Because these functions are in a state machine, the functions will be run multiple times while in their associated state, so we need to control how and when values are updated. The robot then turns without assigning a new goal yaw until it reaches the goal yaw in which it resets the `direction` and `gYaw` variables and assigns the next state to be “Straight.”

The last state “Stop” is a trivial state that simply calls the `stop()` function to set the wheel speeds to 0. As you may have noticed, the `stop()` function is called in other states as well when transitioning states. This is to provide a safety buffer between set wheel speeds so that the robot does not bug out with wheel speeds we did not assign.

If you would like more specific information about how the controller works, both node files are included in the `src` directory of the git hub repository and include comments for each function to describe what it does.

Now that we have our package set up with all the correct modules associated logic, we can run our simulation using Webots and ROS2. To do this, we must first build the package one more time in our base workspace directory using the following command:

```
colcon build --packages-select navigator_package
```

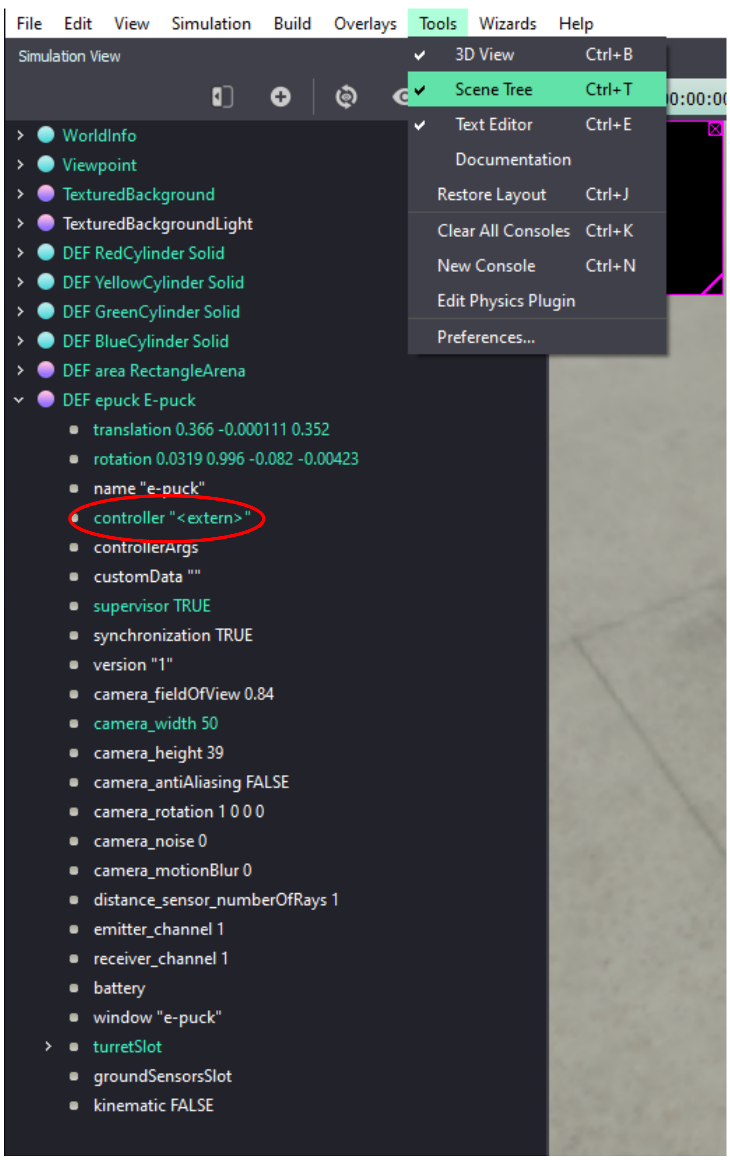
Then we must source our setup script by entering:

```
source install/setup.bash
```

Lastly, we launch the package and simulation by entering the command:

```
ros2 launch navigator_package navigator_launch.py
```

This should launch the simulation in Webots and start the robot according to the controller in the package. If the robot does not start following the controller, you may need to make sure that controller assigned in your Webots simulation is `<extern>`. You can assign this field by opening the Scene Tree and under the DEF epuck E-puck device, assign the “controller” field with `<extern>`.



4. Final Notes and Video:

You should now be able to test and run your custom controller with ROS2 and Webots by using the last three commands shown in the previous chapter. As described earlier the robot should navigate all cells as well as maintain an accurate calculation of its current location as it moves from cell to cell. A video showing the completion of this task is linked below which provides some more information about how the controller operates upon execution and shows the robot in action.

Task 1 Video: https://youtu.be/fbJ_OiZrwXY

You should hopefully also understand how ROS can be used for many different applications outside of Webots through its use of the producer/consumer paradigm and its node networking platform.

6. TODO:

- Fix up documentation by including internal references and links
- Continue to proofread and make more concise
- Add more to final notes

7. References:

Soft Illusion Webots ROS2 Tutorial Videos:
https://youtube.com/playlist?list=PLt69C9MnPchkP0ZXZOqmIGRTOch8o9GiQ

ROS Documentation Homepage:
http://wiki.ros.org/Documentation

Creating Your First ROS2 Package Documentation:
https://docs.ros.org/en/foxy/Tutorials/Creating-Your-First-ROS2-Package.html

Webots Camera Documentation:
https://cyberbotics.com/doc/reference/camera