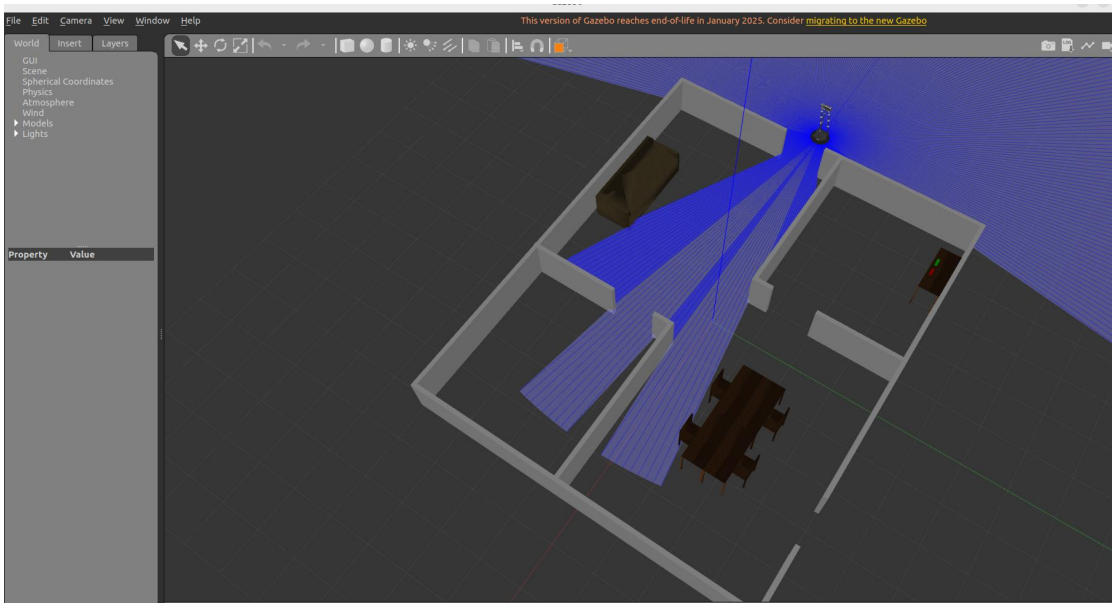
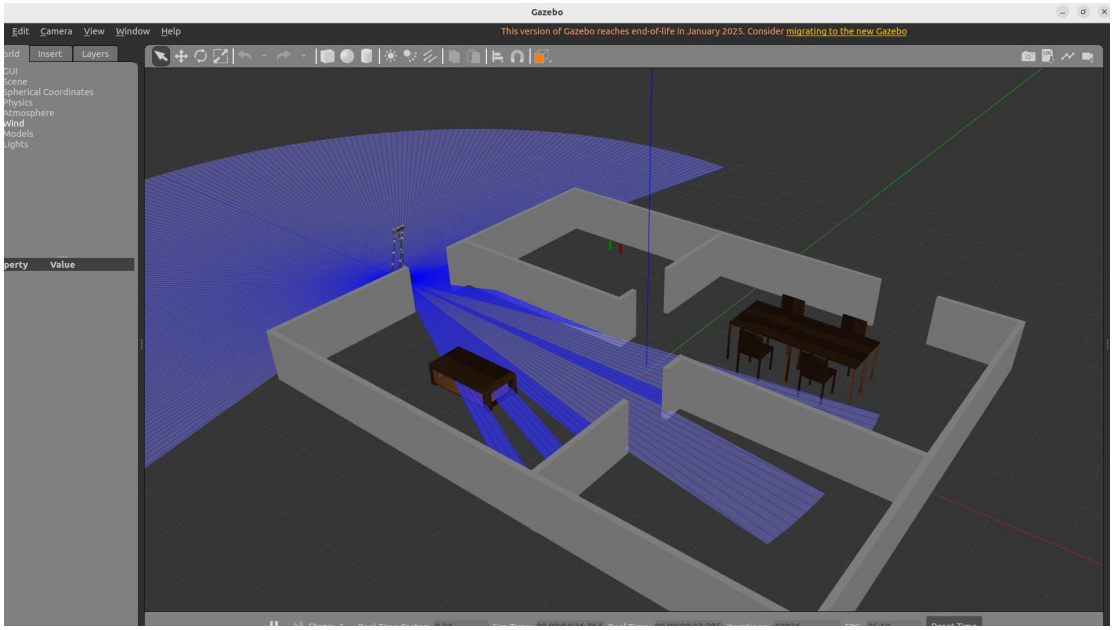
ROS2 Simulation Environment Loading Issue

Problem: When launching the simulation, objects may randomly disappear.

This occurs each time Gazebo is run.



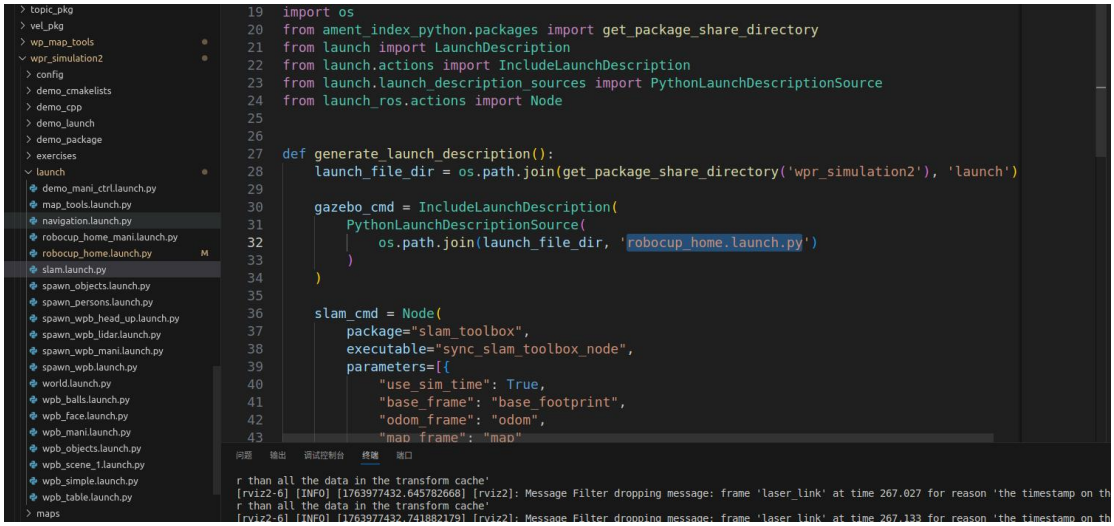
The first launch



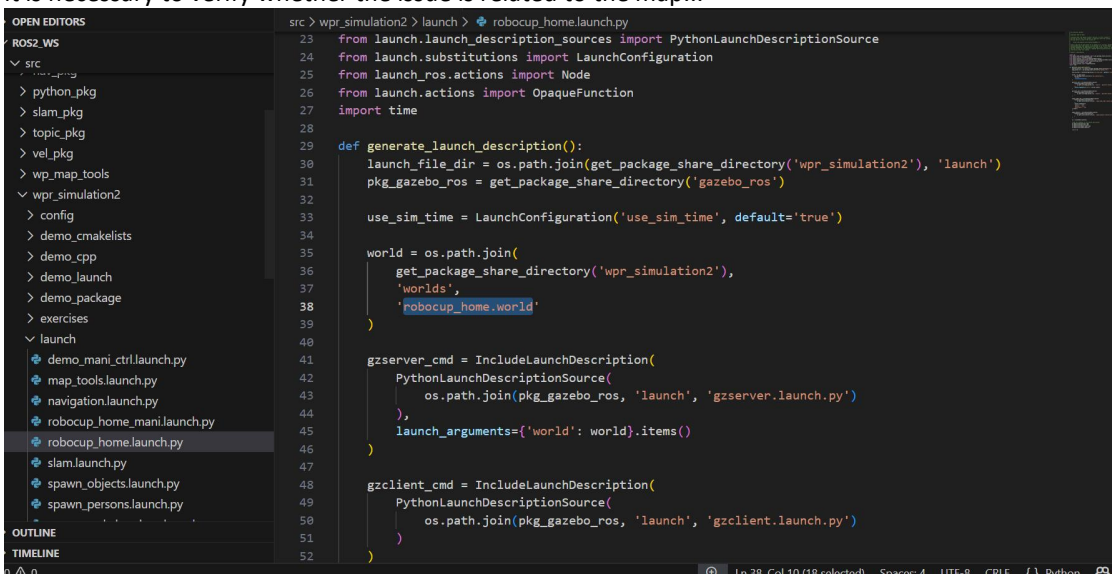
The second launch

This is strange··· further investigation is required...

Inside slam.launch.py, at line 32, this program launches a robocup_home.launch.py file··· continuing the trace...



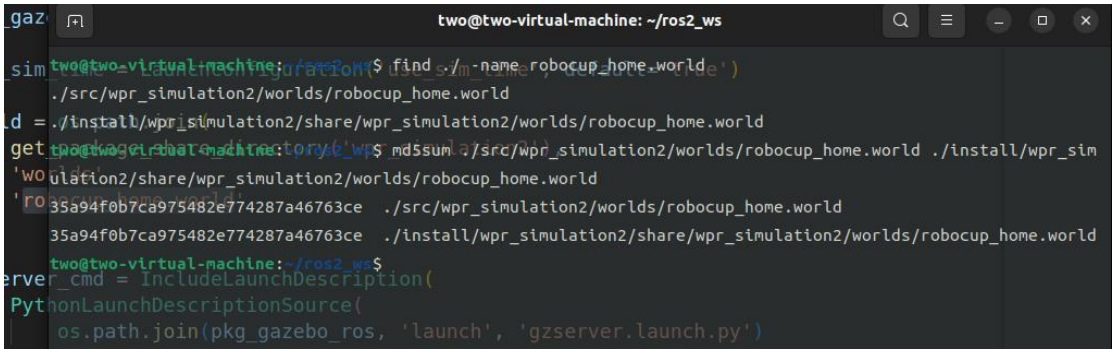
It is necessary to verify whether the issue is related to the map...



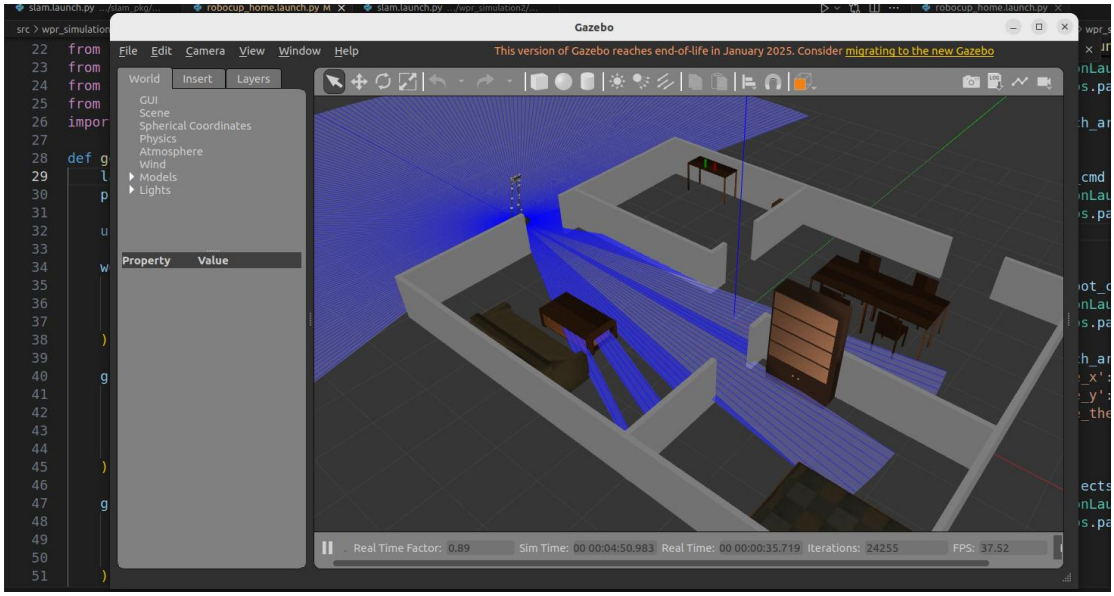
Use the find command to locate the robocup_home.world map...

$ find ./ -name robocup_home.world

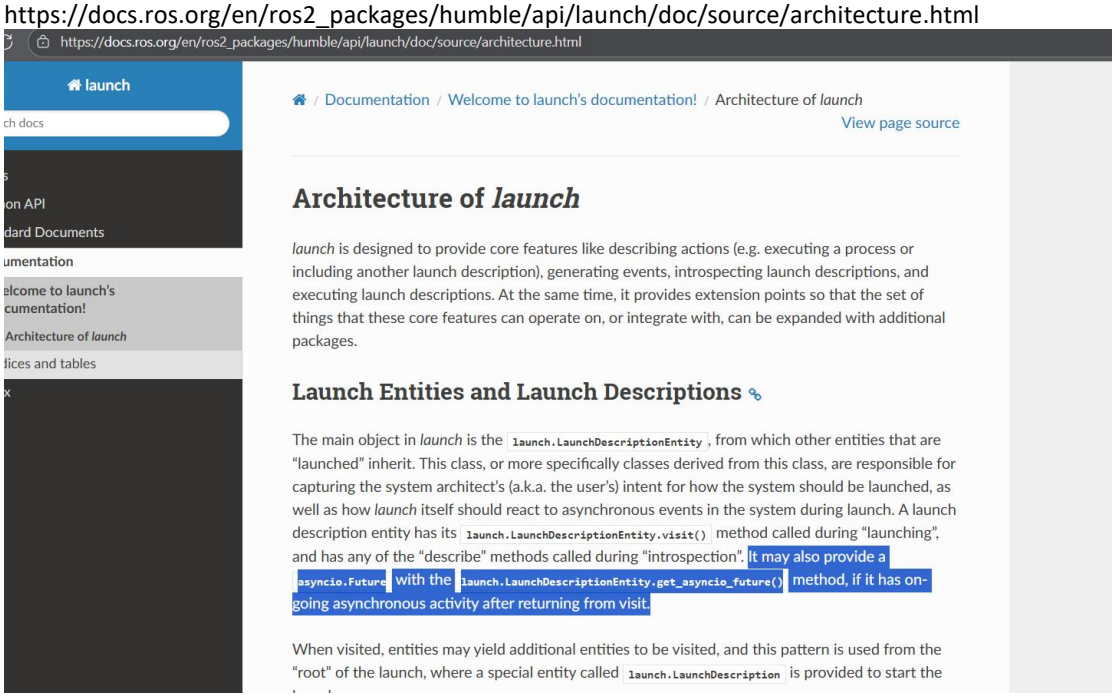
Check the two MD5 values — it's clear that they are unrelated to the map loading error...



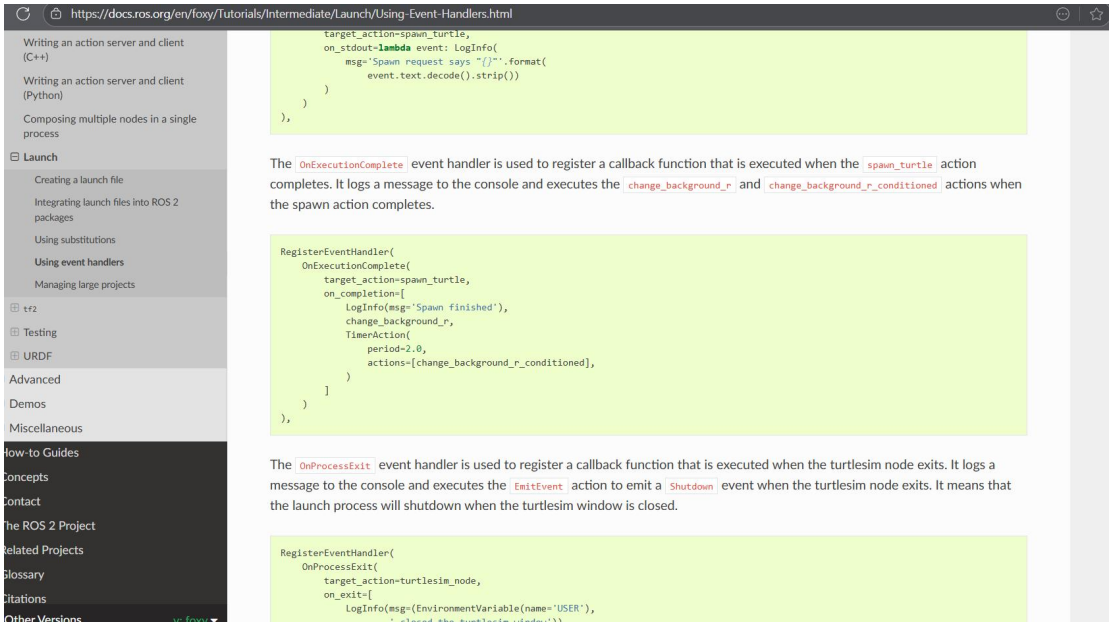
Then, directly run $ ros2 launch wpr_simulation2 robocup_home.launch.py.



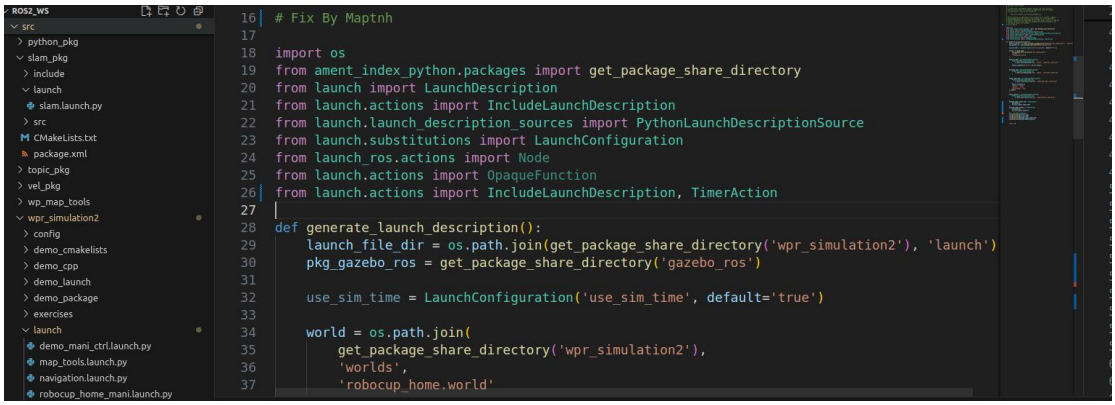
This time there was no response··· I suspect it might be related to memory loading, so I plan to first launch the Gazebo simulation, then delay 20 seconds before loading the components, giving the system enough time to reclaim memory.
Refer to the official documentation:
https://docs.ros.org/en/ros2_packages/humble/api/launch/doc/source/architecture.html



The official ROS2 documentation clearly states that launch files are asynchronous and event-driven. This means that using a standard time delay will not work.
Refer to the TimerAction tutorial: https://docs.ros.org/en/foxy/Tutorials/Intermediate/Launch/Using-Event-Handlers.html



Try modifying robocup_home.launch.py by adding from launch.actions import IncludeLaunchDescription, TimerAction at line 26.。

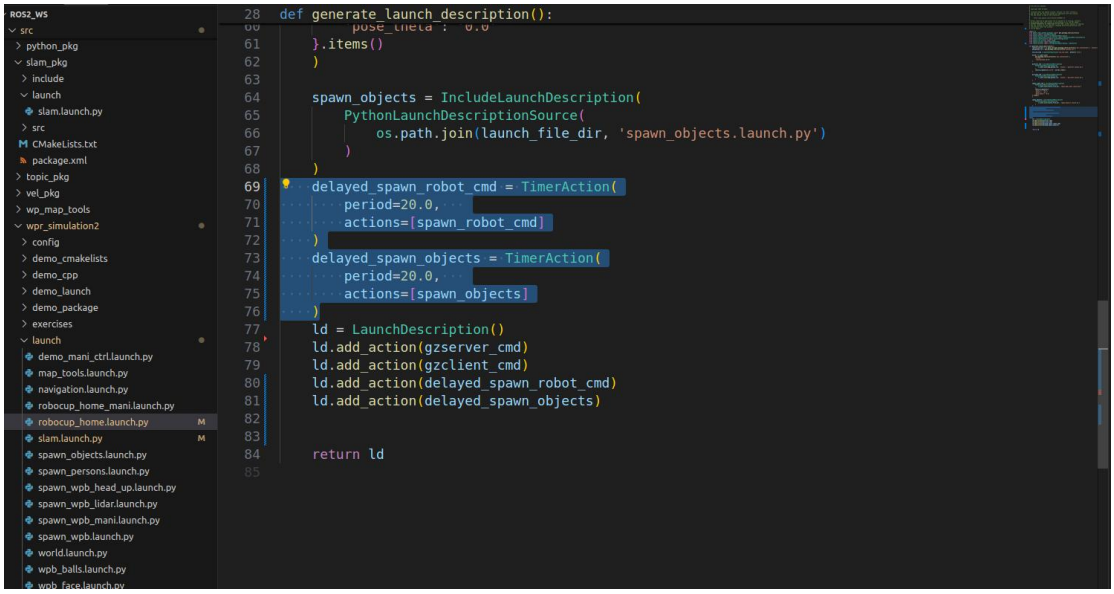


Add the following code...

```
delayed_spawn_robot_cmd = TimerAction(
    period=20.0,
    actions=[spawn_robot_cmd]
)
delayed_spawn_objects = TimerAction(
    period=20.0,
    actions=[spawn_objects]
)
<SNIP>
    ld.add_action(delayed_spawn_robot_cmd)
    ld.add_action(delayed_spawn_objects)
```

Delay the component startup...

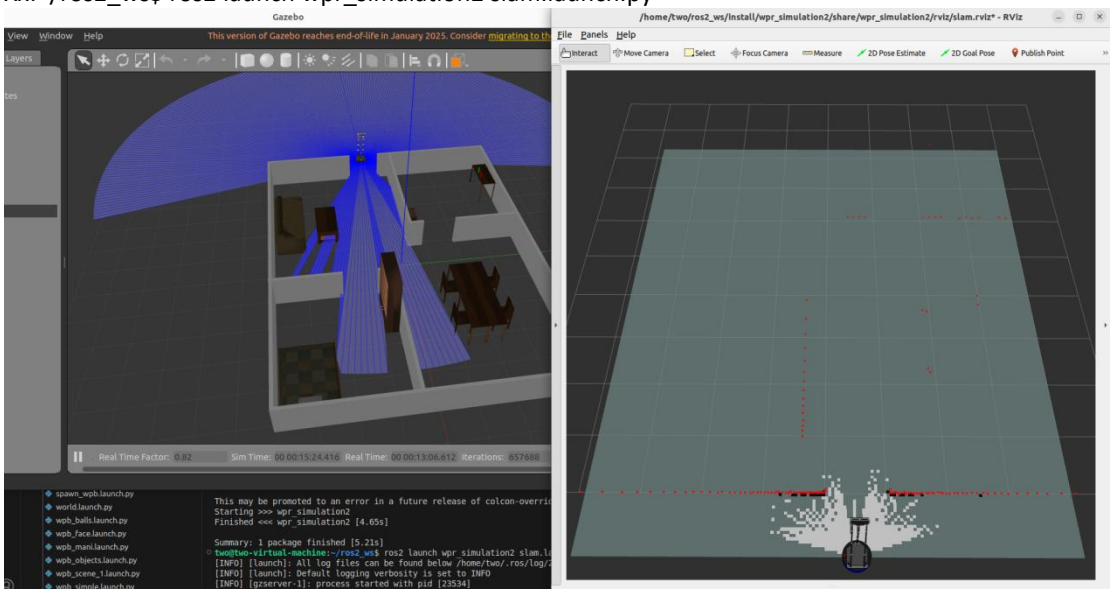
Maptnh@S-H4CK13



Compiling the code...

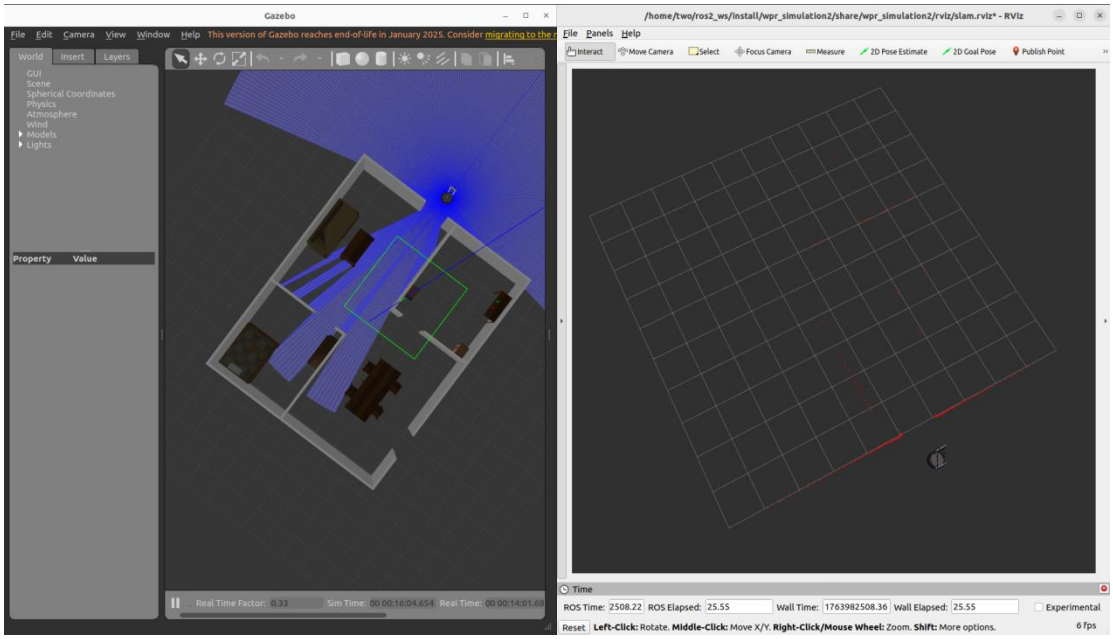
Xx:~/ros2_ws$ colcon build --packages-select wpr_simulation2

Xx:~/ros2_ws$ ros2 launch wpr_simulation2 slam.launch.py

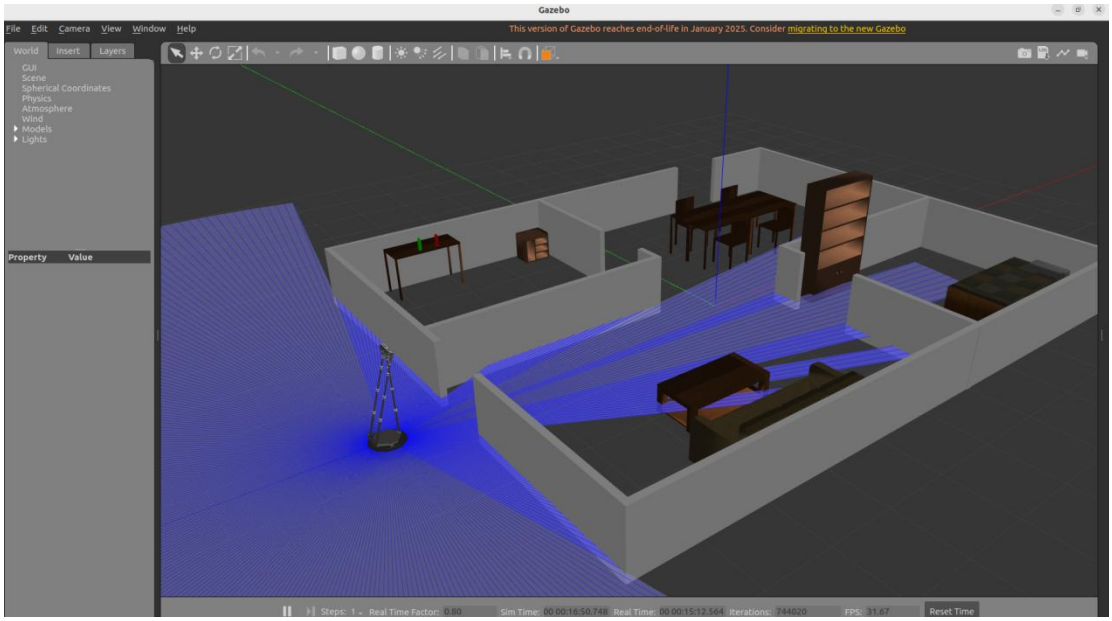


The first launch

Maptnh@S-H4CK13



The second launch



The third launch

```
#!/usr/bin/env python3
# wpr_simulation2/launch/robocup_home.launch.py
# Copyright 2023 6-robot.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Maptnh@S-H4CK13

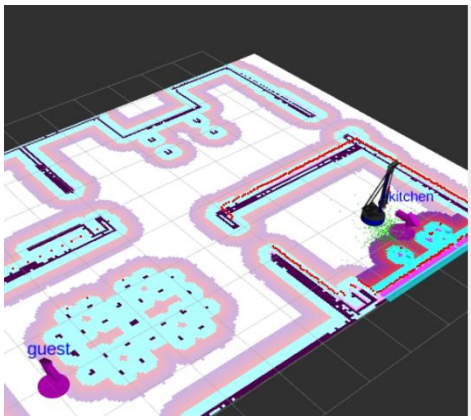
```
# Fix By Maptnh

import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node
from launch.actions import OpaqueFunction
from launch.actions import IncludeLaunchDescription, TimerAction

def generate_launch_description():
    launch_file_dir = os.path.join(get_package_share_directory('wpr_simulation2'), 'launch')
    pkg_gazebo_ros = get_package_share_directory('gazebo_ros')

    use_sim_time = LaunchConfiguration('use_sim_time', default='true')

    world = os.path.join(
        get_package_share_directory('wpr_simulation2'),
        'worlds',
        'robocup_home.world'
    )

    gzserver_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_gazebo_ros, 'launch', 'gzserver.launch.py')
        ),
        launch_arguments={'world': world}.items()
    )

    gzclient_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_gazebo_ros, 'launch', 'gzclient.launch.py')
        )
    )

    spawn_robot_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(launch_file_dir, 'spawn_wpb_lidar.launch.py')
        ),
        launch_arguments={
        'pose_x': '-6.0',
        'pose_y': '-0.5',
        'pose_theta': '0.0'
    }.items()
    )

    spawn_objects = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(launch_file_dir, 'spawn_objects.launch.py')
        )
    )
    delayed_spawn_robot_cmd = TimerAction(
        period=40.0,
        actions=[spawn_robot_cmd]
    )
```

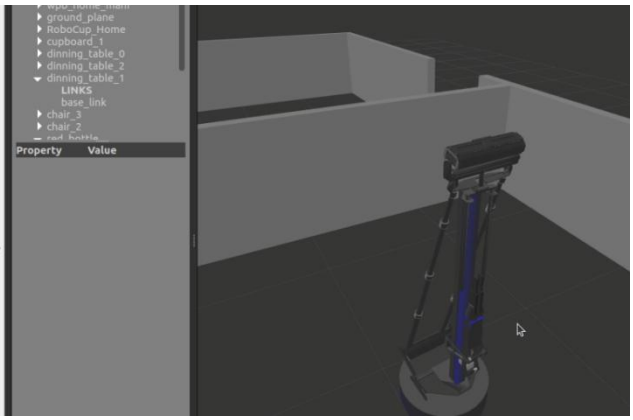
```
    delayed_spawn_objects = TimerAction(
        period=40.0,
        actions=[spawn_objects]
    )
    ld = LaunchDescription()
    ld.add_action(gzserver_cmd)
    ld.add_action(gzclient_cmd)
    ld.add_action(delayed_spawn_robot_cmd)
    ld.add_action(delayed_spawn_objects)


    return ld
--------------------------------------------------
```

The original implementation overlooked that starting numerous programs at once could immediately trigger memory overflow, resulting in system failure.

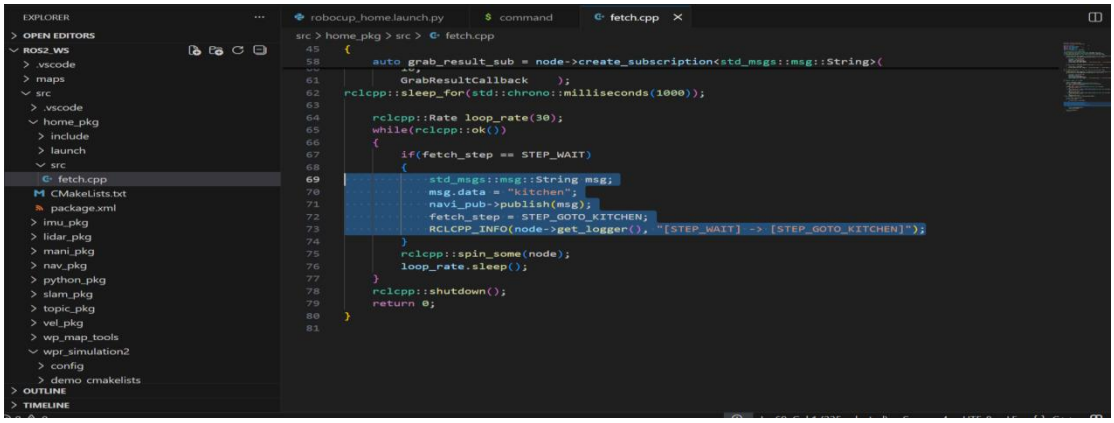
ROS2 Robot Handling Program: Reverse Analysis and Control Hijack Bypass

There is no denying that this part is quite complex; I also spent a long time analyzing how it works... The first time I used the official command $ ros2 run home_pkg fetch, the robot seemed unable to perform the grasp, and inexplicably moved backward...





Open fetch.cpp



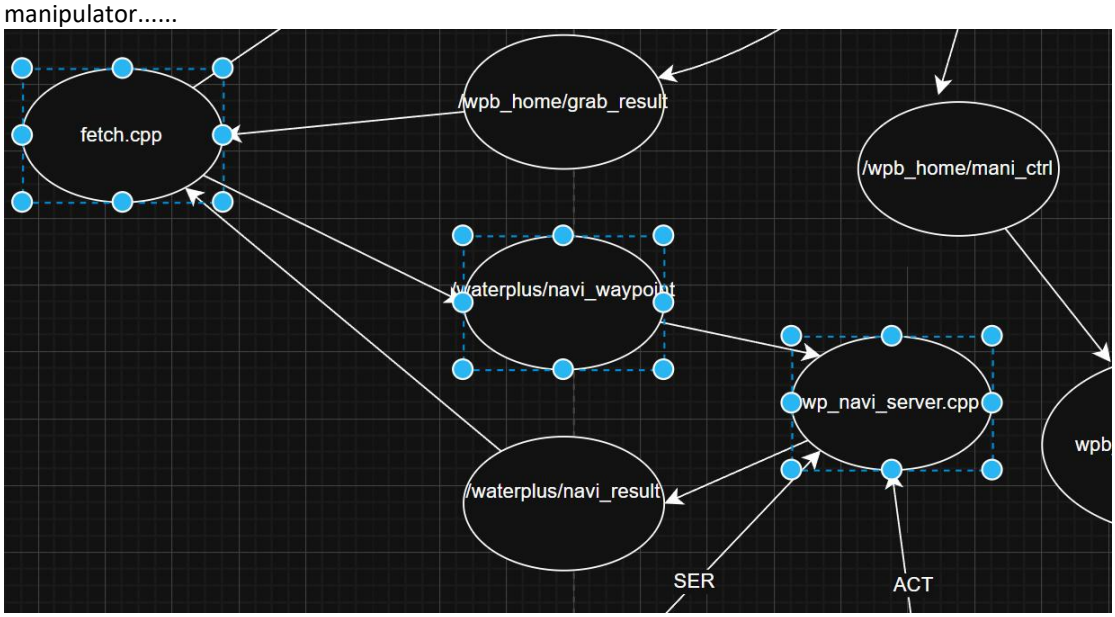
At line 69, the code sends the navigation coordinates to the navi_pub node...

```
}
int main(int argc, char** argv)
{
    rclcpp::init(argc, argv);
    node = std::make_shared<rclcpp::Node>("fetch_node");
    navi_pub = node->create_publisher<std_msgs::msg::String>(
        "/waterplus/navi_waypoint",
        10    );
    behavior_pub = node->create_publisher<std_msgs::msg::String>(
        "/wpb_home/behavior",
        10    );
    auto navi_result_sub = node->create_subscription<std_msgs::msg::String>(
        "waterplus/navi_result",
        10,
        NaviResultCallback    );
    auto grab_result_sub = node->create_subscription<std_msgs::msg::String>(
        "/wpb_home/grab_result",
        10,
        GrabResultCallback    );
```

We need to find its actual handler.....

I spent a lot of time mapping out their relationships, which looks like this...

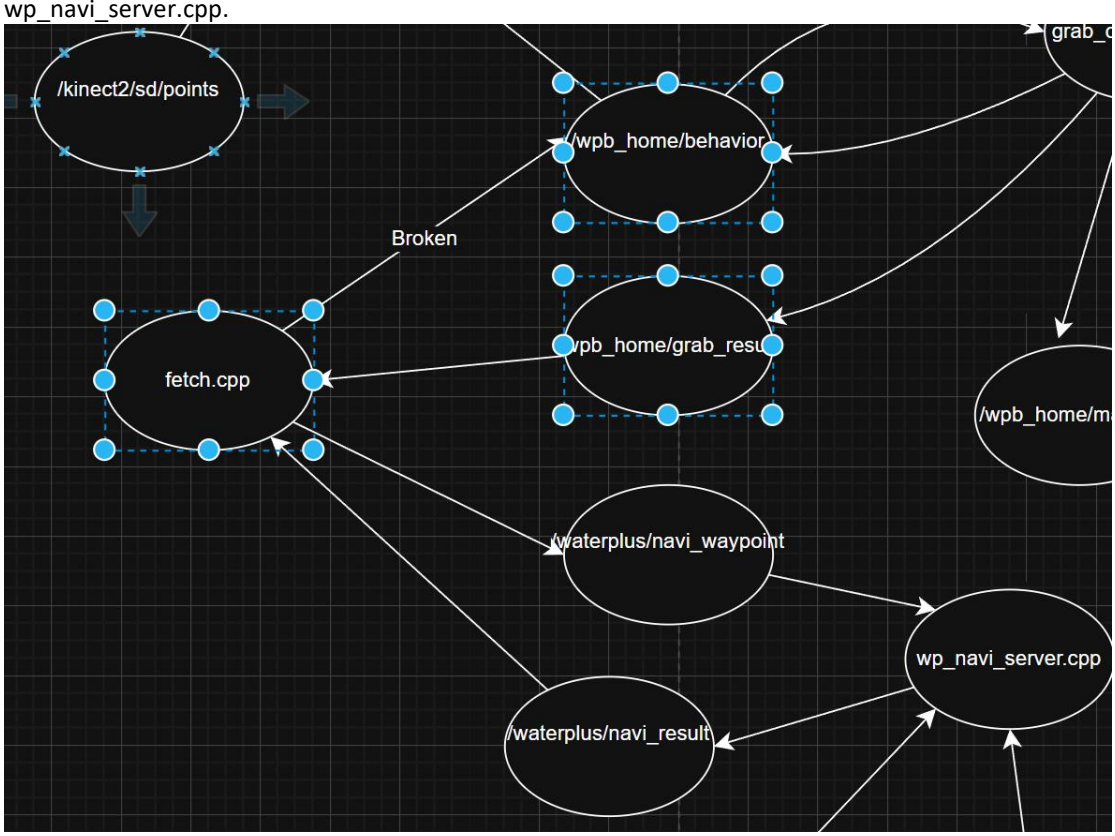
We need to follow the chain to locate the program that controls the manipulator.....



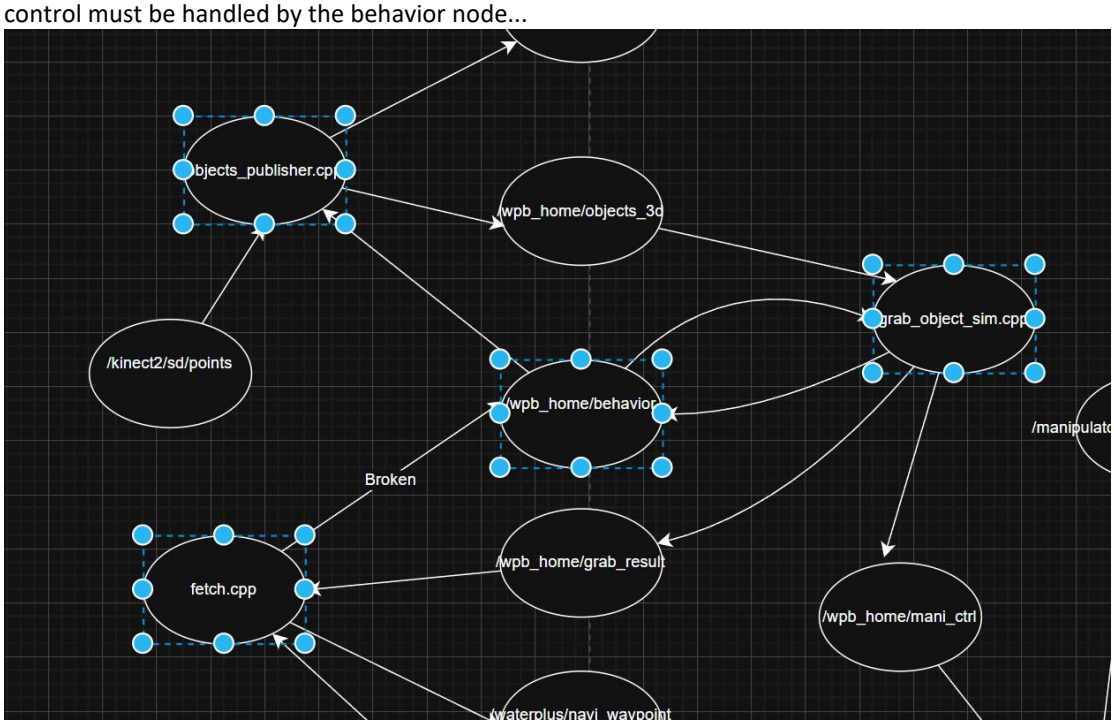
Upon receiving the execution point name, it will attempt to navigate.....

The data is then returned to fetch.cpp for the next grasping operation.

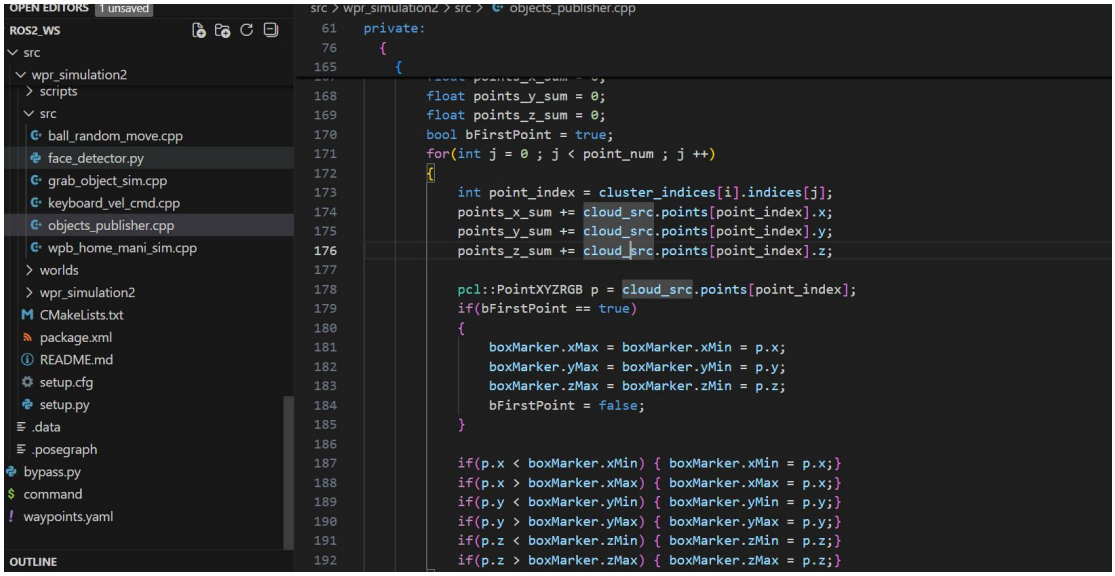
Therefore, our control target is not in wp_navi_server.cpp.



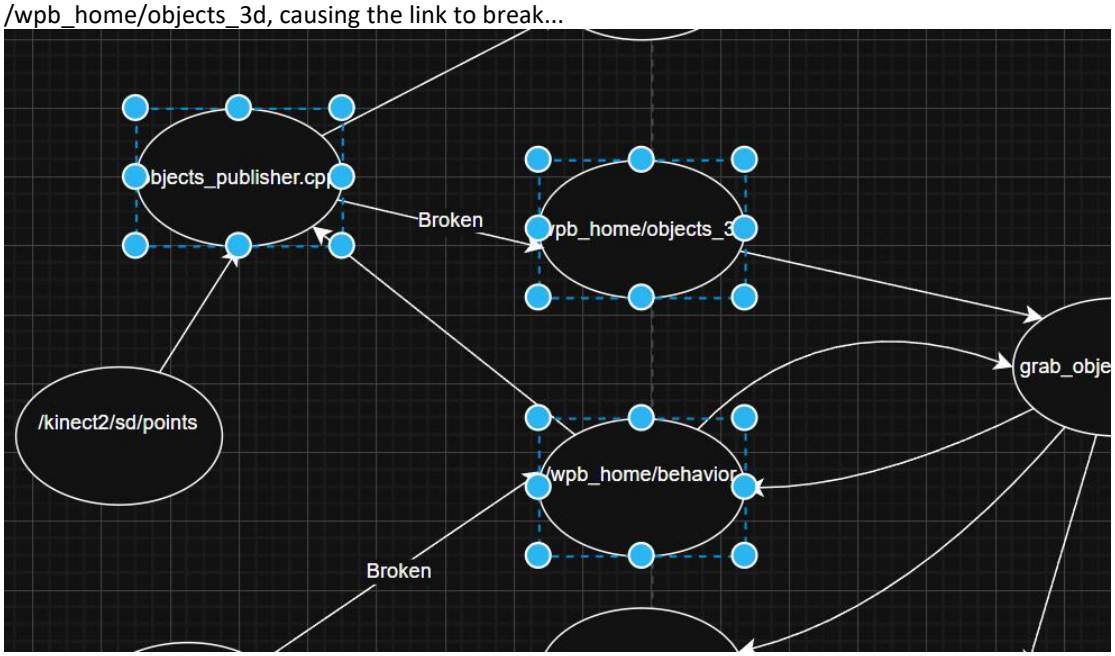
By excluding the navi node in fetch.cpp, we are left with the grab_result and behavior nodes. Further excluding grab_result (which typically only receives results), it is clear that fetch's grasping control must be handled by the behavior node...



The /wpb_home/behavior node is linked to objects_publisher.cpp and grab_object_sim.cpp. objects_publisher.cpp handles the recognition of object point clouds.

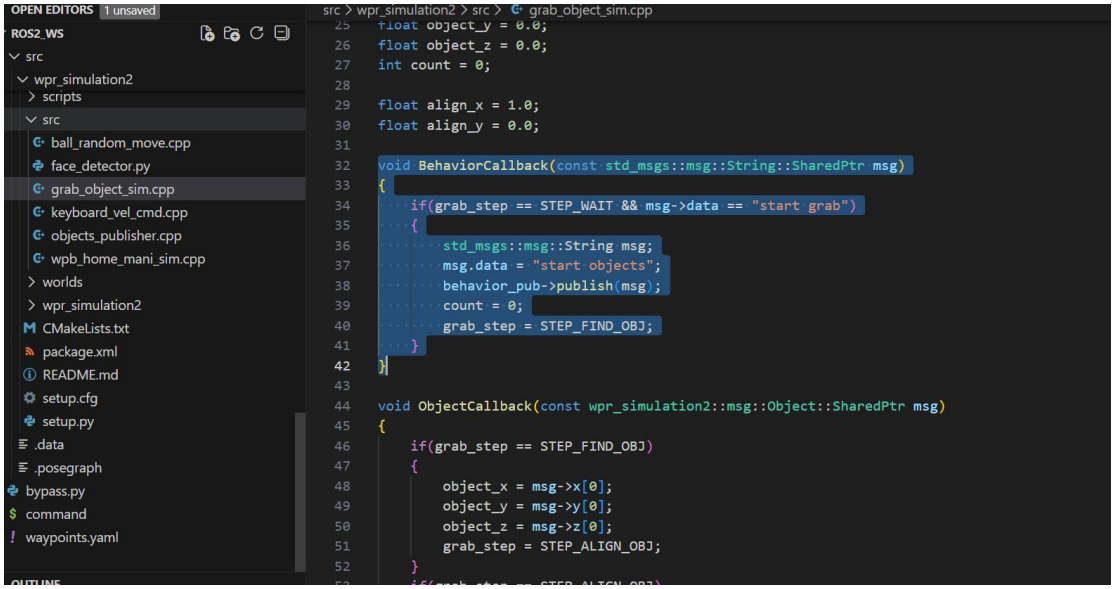


Failure to detect the object prevents the coordinate data from being transmitted to /wpb_home/objects_3d, causing the link to break...

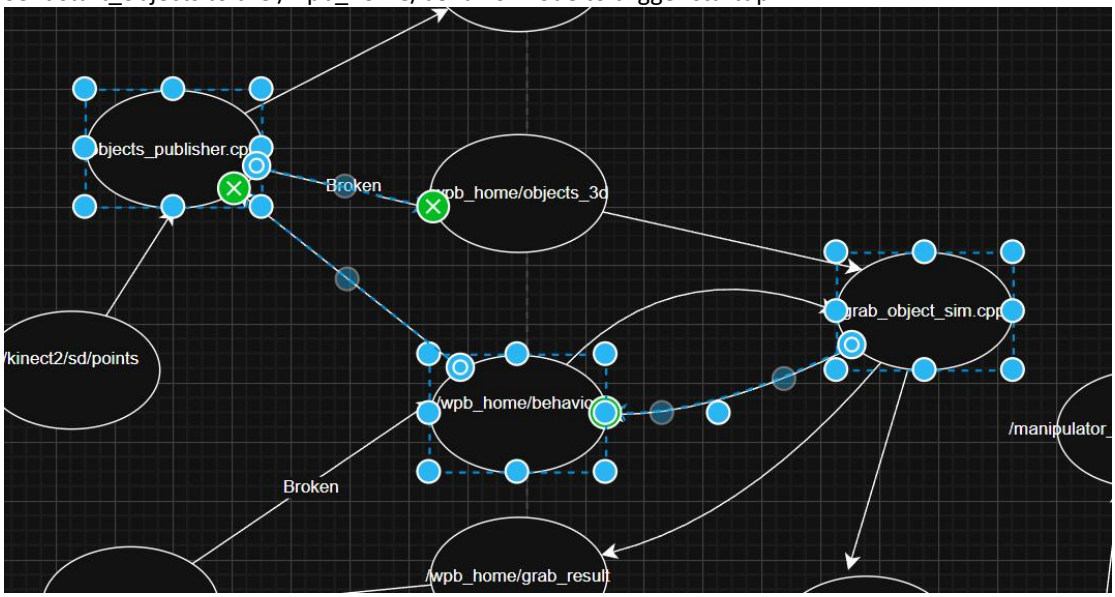


Follow through to grab_object_sim.cpp.

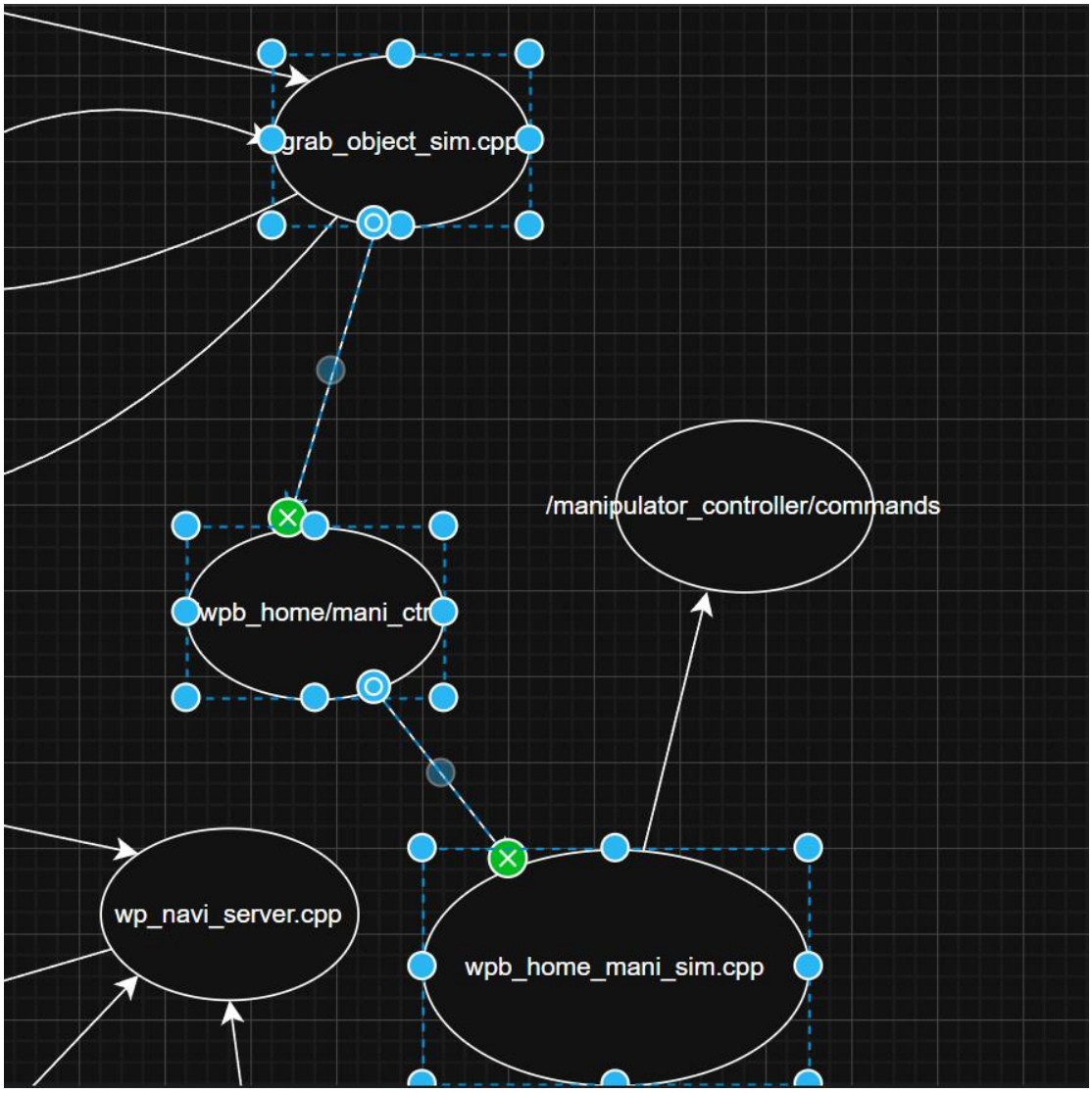
Maptnh@S-H4CK13



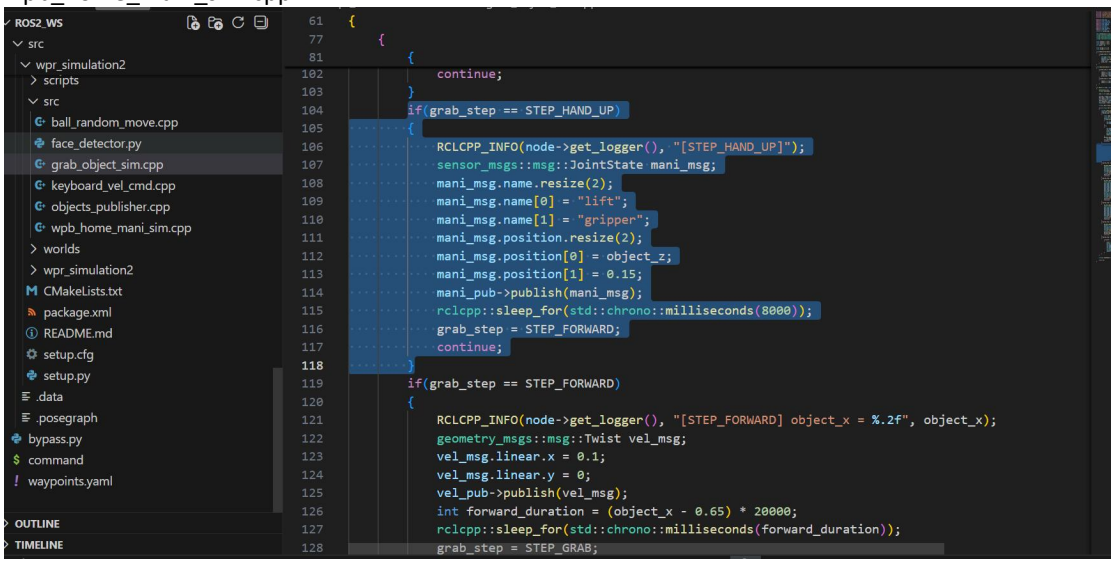
Sent start_objects to the /wpb_home/behavior node to trigger startup.



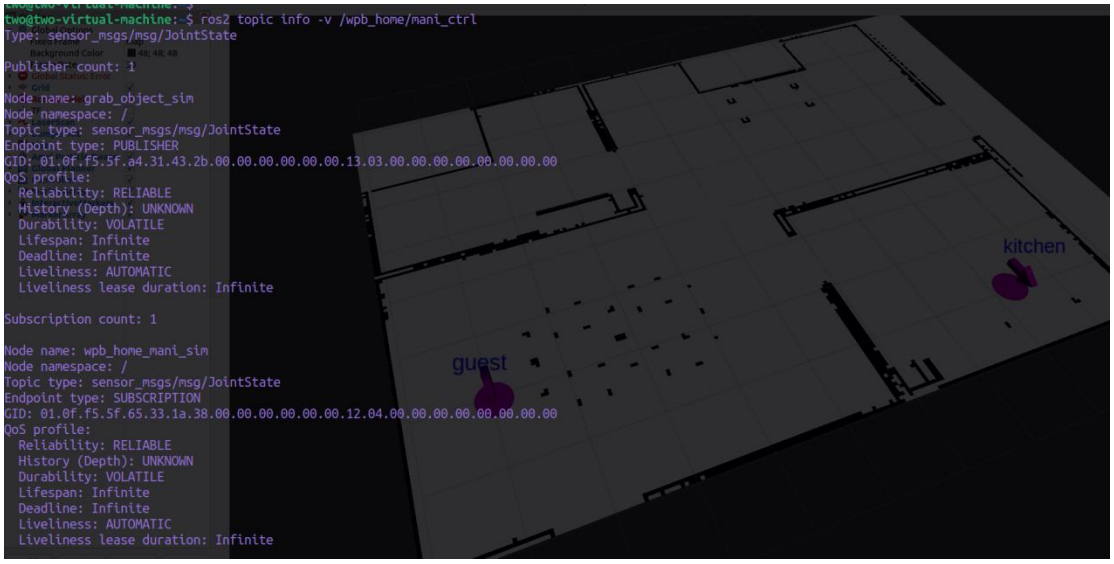
In reality, this approach is ineffective, as the object coordinates cannot be retrieved...



Follow through to
wpb_home_mani_sim.cpp.

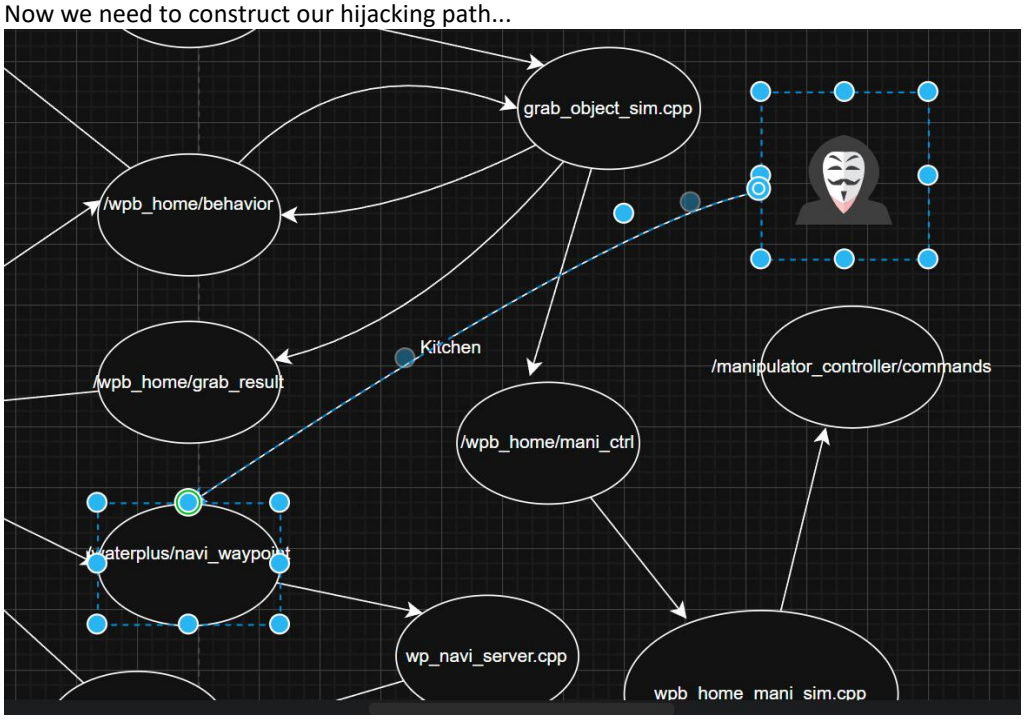


Fortunately, we located the manipulator control logic...
Now we inspect the node details.
$ ros2 topic info -v /wpb_home/mani_ctrl

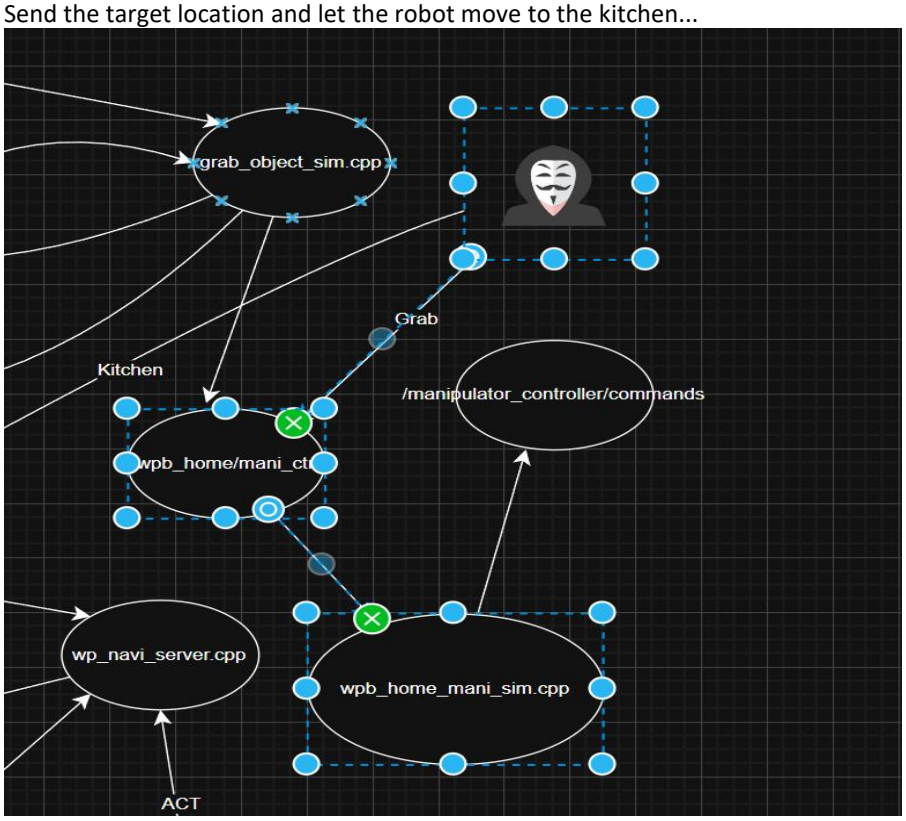


1. name: string[]
The names of all robot joints.
Length = number of joints.
Example: ["shoulder_pan", "shoulder_lift", "elbow", ...]
You can find the corresponding joint names in grab_object_sim.cpp (lines 76–191).
2. position: float64[]
The angles or linear displacements of each joint.
Units: usually radians or meters (for linear sliders).
Length must match name[].
Do not leave empty → this will cause errors or the robot won't move.
3. velocity: float64[]
The velocity of each joint (rad/s or m/s).
Can be left empty → use an empty array if not needed.
4. effort: float64[]
The torque or force of each joint (N·m or N).
Usually not needed in simulation → can be an empty array.

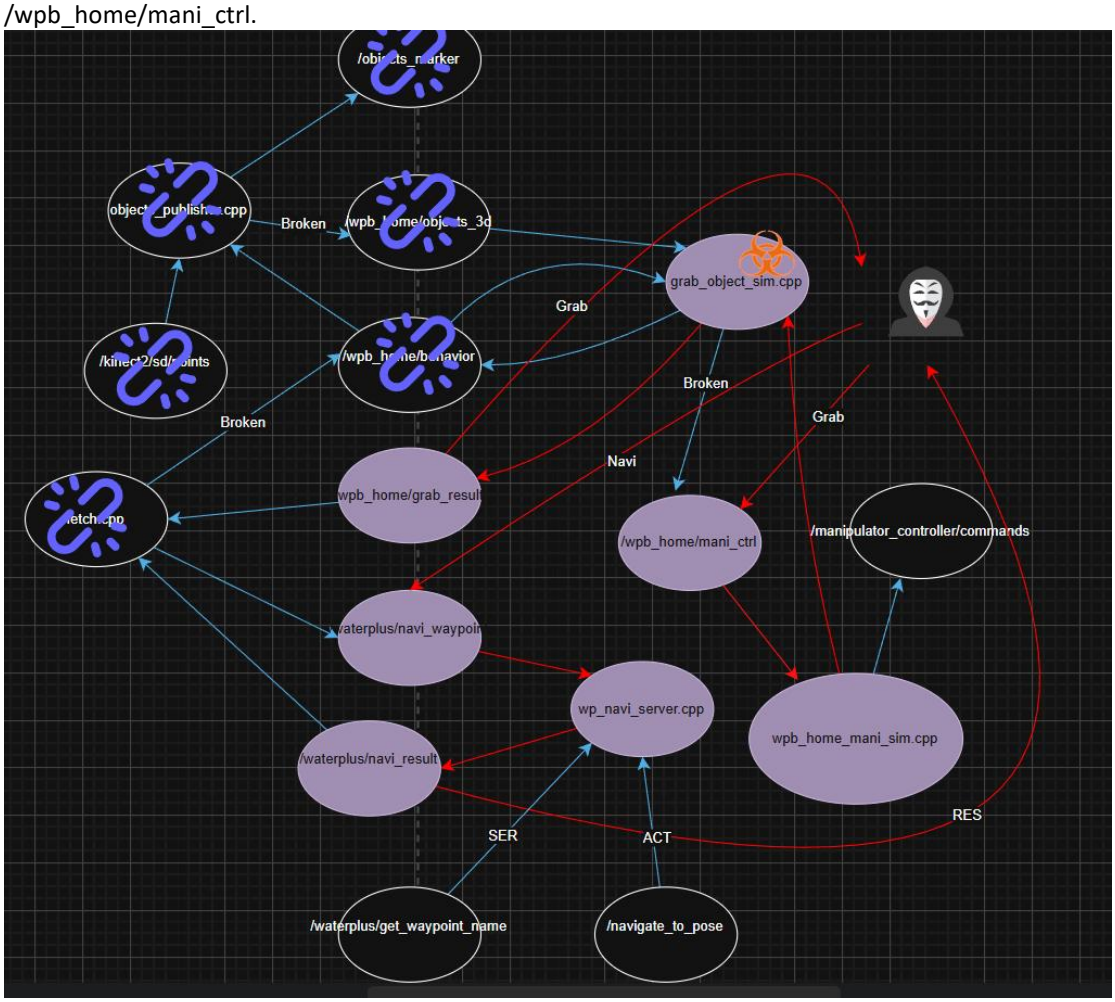
Now we need to construct our hijacking path...



Send the target location and let the robot move to the kitchen...



Send control data directly to /wpb_home/mani_ctrl.



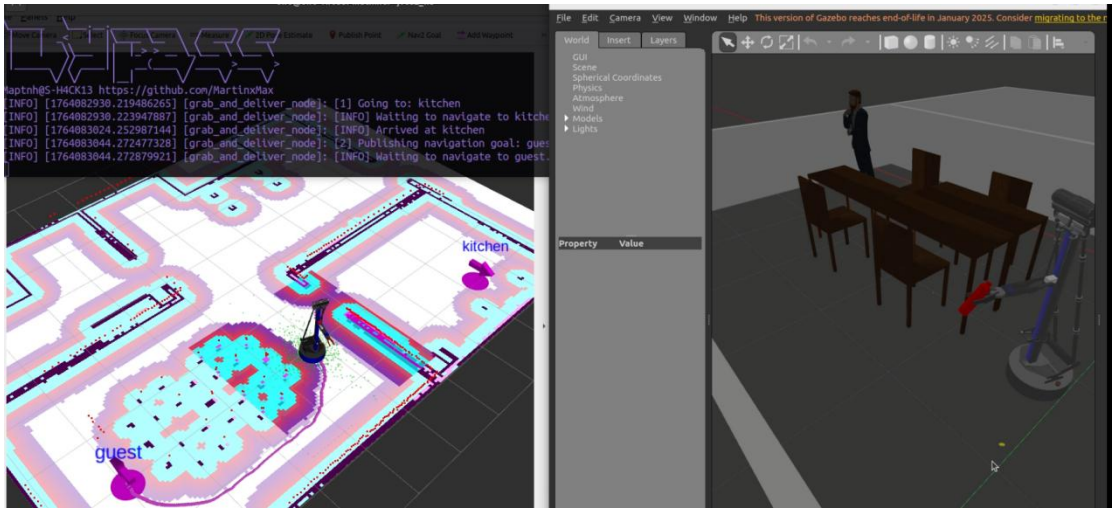
A complete hijacking chain....executed through scripts.

```
#!/usr/bin/env python3
# Maptnh
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
from sensor_msgs.msg import JointState
from geometry_msgs.msg import Twist
import time

LOGO = '''
__________
\______   \____.__.___________   ______ ______
 |    |  _<  |  |\____ \__  \ /  ___//  ___/
 |    |   \\___  ||  |_> > __ \_\___ \ \___ \
 |______  // ____||   __(____  /____  >____  >
        \/ \/     |__|       \/     \/     \/
Maptnh@S-H4CK13 https://github.com/MartinxMax'''

class GrabAndDeliver(Node):
    def __init__(self):
        super().__init__('grab_and_deliver_node')
        self.target_lift = 0.80
        self.gripper_open = 0.15
        self.gripper_close = 0.07
        self.lift_up_delta = 0.05
        self.forward_duration = 3.0
        self.backward_duration = 3.0
        self.sleep_short = 1.0
        self.sleep_long = 4.0
        self.navi_pub = self.create_publisher(String, '/waterplus/navi_waypoint', 10)
        self.mani_pub = self.create_publisher(JointState, '/wpb_home/mani_ctrl', 10)
        self.cmd_vel_pub = self.create_publisher(Twist, '/cmd_vel', 10)
        self.navi_done = False
        self.sub = self.create_subscription(String, '/waterplus/navi_result', self.navi_callback, 10)

    def navi_callback(self, msg):
        if msg.data == 'navi done':
            self.navi_done = True

    def wait_for_navi_done(self, target_name):
        self.get_logger().info(f'[INFO] Waiting to navigate to {target_name}...')
        while rclpy.ok() and not self.navi_done:
            rclpy.spin_once(self, timeout_sec=0.1)
        self.navi_done = False
        self.get_logger().info(f'[INFO] Arrived at {target_name}')

    def publish_mani(self, lift, gripper):
        msg = JointState()
        msg.name = ['lift', 'gripper']
        msg.position = [lift, gripper]
        self.mani_pub.publish(msg)

    def publish_cmd_vel(self, linear_x):
        msg = Twist()
```

```
        msg.linear.x = linear_x
        msg.linear.y = 0.0
        msg.linear.z = 0.0
        msg.angular.x = 0.0
        msg.angular.y = 0.0
        msg.angular.z = 0.0
        self.cmd_vel_pub.publish(msg)

    def run_sequence(self):
        self.get_logger().info('[1] Publishing navigation goal: kitchen')
        self.navi_pub.publish(String(data='kitchen'))
        self.wait_for_navi_done('kitchen')

        self.publish_mani(self.target_lift, self.gripper_open)
        time.sleep(self.sleep_long)

        self.publish_cmd_vel(0.1)
        time.sleep(self.forward_duration)

        self.publish_cmd_vel(0.0)
        time.sleep(self.sleep_short)

        self.publish_mani(self.target_lift, self.gripper_close)
        time.sleep(self.sleep_long)

        target_lift_up = self.target_lift + self.lift_up_delta
        self.publish_mani(target_lift_up, self.gripper_close)
        time.sleep(self.sleep_long)

        self.publish_cmd_vel(-0.1)
        time.sleep(self.backward_duration)

        self.publish_cmd_vel(0.0)
        time.sleep(self.sleep_short)

        self.get_logger().info('[2] Publishing navigation goal: guest')
        self.navi_pub.publish(String(data='guest'))
        self.wait_for_navi_done('guest')

        self.publish_mani(target_lift_up, self.gripper_open)
        time.sleep(self.sleep_long)

        self.get_logger().info('=== Sequence Completed ===')

def main(args=None):
    print(LOGO)
    rclpy.init(args=args)
    node = GrabAndDeliver()
    node.run_sequence()
    node.destroy_node()
    rclpy.shutdown()

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

```
$ source install/setup.bash;ros2 launch home_pkg home.launch.py
$ python3 bypass.py
```

The parameters in the script are all pre-calculated. For grasping objects in different positions, you usually need to adjust the waypoints.yaml accordingly. The object's position should align with the robot's X-
axis.



It started to behave exactly as we scripted it...